17588

NATIONAL LIBRARY
OTTAWA

CANADA

BIBLIOTHÈQUE NATIONALE
OTTAWA

NAME OF AUTHOR. DONALD PAUL LAFFIN

TITLE OF THESIS. DESIGN AND IMPLEMENTATION OF DECISION-TABLE LANGUAGES

UNIVERSITY. OF ALBERTA

DEGREE FOR WHICH THESIS WAS PRESENTED. M.Sc

YEAR THIS DEGREE GRANTED. 1973

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed). D. Paul Laffin

PERMANENT ADDRESS:

K394 HARVARD AVE
MONTREAL 260
QUEBEC

DATED. Aug. 24, 1973.

NL-91 (10-68)

THE UNIVERSITY OF ALBERTA

DESIGN AND IMPLEMENTATION OF DECISION-TABLE LANGUAGES

by

© DONALD PAUL LAFFIN

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1973

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "Design and Implementation of Decision-Table Languages", submitted by Donald Paul Laffin in partial fulfilment of the requirements for the degree of Master of Science.

Barry J. Mailloux
Supervisor

Date Aug. 20, 1973

ABSTRACT

Decision tables can be used to describe the logical structure of a procedure. The purpose of this study is to investigate the problems involved in the design and implementation of programming languages based on decision tables.

A decision-table language is developed during an analysis of language design problems. Specific methods are proposed for the implementation of a number of features of the language. Several techniques for the generation of code from decision tables are reviewed and illustrated with examples. Reference is made to DET, which is a decision-table language designed and implemented as part of this research.

ACKNOWLEDGMENTS

The author wishes to extend special thanks to his supervisor, Dr. B. J. Mailloux, for his guidance and assistance in this research.

Financial support from the Department of Computing Science, in the form of Graduate Teaching Assistantships, and from the National Research Council, in the form of a Postgraduate Scholarship, is gratefully acknowledged.

Finally, the author expresses great appreciation for the support and encouragement of his wife, Mary, whose many hours of typing and proofreading have made the preparation of this thesis much easier.

TABLE OF CONTENTS

CHAPTER		PAGE
1.	Introduction	1
2.	What Are Decision Tables?	3
	2.1. Basic Features	3
	2.2. History and Development	11
3.	Variations on Table Layout	16
	3.1. Extended Answer Tables	16
	3.2. Else Answer	18
	3.3. Ordering Within Rules	21
	3.4. Compound Questions	22
	3.5. OR Tables	24
	3.6. Horizontal Tables	25
	3.7. Extended Actions	26
	3.8. Linkage	28
	3.8.1. Table Identification	28
	3.8.2. Table Initialization	29
	3.8.3. Open or Closed Tables	30
	3.8.4. Flow Sructure	30
	3.8.5. Modular Structure	31
	3.8.6. External Structure	33
4.	Are Decision Tables Better?	34
	4.1. Advantages and Disadvantages	34
	4.1.1. Narratives	34

TABLE OF CONTENTS (continued)

CHAPTER PAGE

4.1.2. Flowcharts 35

4.1.3. Decision Tables 37

4.2. Why Doesn't Everyone Use Decision Tables? .. 39

5. Language Design 44

5.1. Design Approach 44

5.2. Table Components 47

5.2.1. Questions 47

5.2.2. Actions 51

5.2.3. Answers 53

5.2.4. Mapping 60

5.3. Joining the Components 62

5.3.1. QUESTION/ANSWER Statement 62

5.3.2. ACTION/MAPPING Statement 65

5.4. Forming a Complete Table 66

5.5. A Complete Program 70

5.6. Limited Tables 73

6. Processor Design 75

6.1. Design Approach 75

6.2. Program Listing 80

6.3. Debugging Aids 83

6.4. Statement Numbers and Variables 86

6.5. Nesting DO-loops and Tables 87

TABLE OF CONTENTS (continued)

CHAPTER PAGE

6.6. Closed Tables 89

6.6.1. Recognition 90

6.6.2. Errors 91

6.6.3. How to Exit 93

6.7. Code for PERFORM and EXIT 94

6.7.1. Computed GOTO Code 96

6.7.2. Assigned GOTO Code 99

6.8. Detecting Errors 102

6.8.1. Preliminary Checks 105

6.8.2. Ambiguity and Completeness 108

6.8.3. Semantic Errors 113

6.9. DET 116

7. Code Generation 118

7.1. Techniques 119

7.1.1. Repetitive Testing 119

7.1.2. Tree 123

7.1.3. Rule Mask 128

7.1.4. Branch Table 131

7.2. Side Effects 133

8. Conclusion 135

Bibliography 138

Appendix: DET Reference Manual 142

TABLE OF CONTENTS (continued)

CHAPTER PAGE

A.1. New Statements 142
A.1.1. TABLE 142
A.1.2. QUESTION/ANSWER 143
A.1.3. ACTIONS 144
A.1.4. ACTION/MAPPING 144
A.1.5. EXIT 147
A.2. Sample Program 148
A.3. Summary of Rules 149
A.4. Use of the Preprocessor 151

LIST OF FIGURES

FIGURE		PAGE
2-1.	Decision table components	3
2-2.	Traditional table names	4
2-3.	Simple table form	5
2-4.	Table for choice of apparel	6
2-5.	Table with redundancy	6
2-6.	Table with don't-care	8
2-7.	Table with hidden redundancy	8
2-8.	Table after reordering	9
2-9.	Extended answer table	10
3-1.	Limited answer table	17
3-2.	Complete extended answer question	19
3-3.	Table with ELSE answer	19
3-4a.	Sequential actions	21
3-4b.	Nonsequential actions	21
3-5a.	Simple questions	23
3-5b.	Compound questions	23
3-6.	OR table	24
3-7.	Horizontal table	26
3-8.	Extended action table	27
5-1.	No else answer	57
5-2.	Global else answer	57
5-3.	Local else answers	58

LIST OF FIGURES (continued)

FIGURE PAGE

5-4. Global and local else answers 58

5-5. Hierarchy of local else answers 59

5-6. A complete table 69

6-1. Listing layout for a limited table 82

CHAPTER 1

Introduction

Decision tables can be used to describe the logical structure of a procedure. The representation of a procedure consists of actions and the decisions which determine what action is to be taken in each possible circumstance. Decision tables were devised as an alternative to narratives and flowcharts. Each of these well established techniques is recognized as being both a means of communication between men, and a form of computer-program documentation. It will be argued that decision tables are superior in each area.

The primary purpose of this study is to investigate the use of decision tables in a third area, that of communication between man and computer. Since a computer is unable to accomplish any useful work until someone has provided a complete and detailed specification of the procedure to be performed, this man-machine communication is of considerable importance.

Since the late 1950's, high-level programming languages have become accepted as the usual technique for the specification of procedures for computer processing. The approach of this study will be to treat decision tables as the central control structure on which a programming

language is based. Control structures for programming languages have been the subject of growing interest since 1968 when a letter from Dijkstra was published in the Communications of the ACM [1]. He argued that elimination of the GOTO statement from programming languages would have a beneficial effect on the writing, reading, and debugging of programs. This is the origin of the "GOTO controversy", which has led to the investigation of alternative control structures [2]. It is not the intent of this study to join in the debate on the merits of other control structures, but rather to concentrate on the development of decision tables.

In order to provide an introduction to the subject, the next three chapters include a description of the basic layout of decision tables, a review of their brief history, an explanation of a number of modifications and extensions, and an evaluation of their advantages and disadvantages. In Chapter 5, problems involved in designing a programming language based on decision tables are discussed. The result of this discussion is the specification of a complete language. Chapters 6 and 7 are concerned with the implementation of a processor to translate decision-table language code into lower-level code. Chapter 7 is devoted to a critical appraisal of research on techniques for generating code from decision tables. The Appendix contains the reference manual for DET, a decision-table language developed during this research.

CHAPTER 2

What are Decision Tables?

2.1 Basic Features

As was said in the Introduction, decision tables can be used to describe the logical structure of a procedure. Fig. 2-1 shows the four sections of a decision table, each of which is named to indicate its contents. The

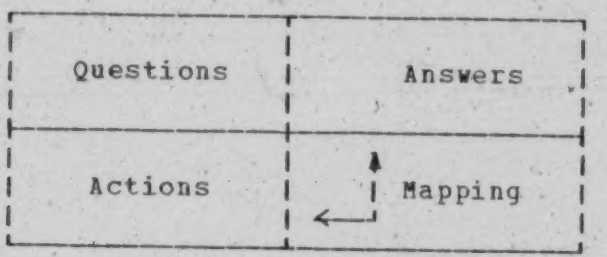

Fig. 2-1 Decision table components.

"questions" are the decisions which must be made in order to determine which "actions", if any, to perform. The "answers" to the questions show all the possible states which can result from the decisions. Finally, the "mapping" indicates which actions are to be performed for each state.

A less desirable convention which has traditionally been used to refer to these four divisions of a decision table, is shown in Fig. 2-2. Since the terms "stub" and "entry" have no apparent mnemonic value, they are

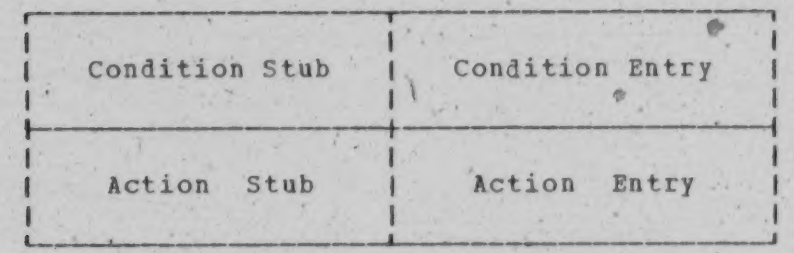

Fig. 2-2 Traditional table names.

easily confused. Their use will be avoided in this thesis.

Decision tables are useful in at least three areas: first, when analyzing an existing procedure, or synthesizing a new one, decision tables can be used to record the logic involved, and to communicate it to others; second, for expressing a procedure in a language which can readily be compiled into an executable computer program; and third, for documenting the logic involved in computer programs. Decision tables allow the precise and unambiguous specification of numerous and interacting decisions, in an intelligible, compact, and tabular form.

Despite general agreement on the purpose of each of the four sections, there is no widely accepted standard which proposes a precise format for each section. A number of factors which are involved in designing a table format will be discussed in later chapters. However, to illustrate this explanation, consider the format used in Fig. 2-3. This format will be used for a number of examples on the following pages. Double lines delineate the four sections

of the table. The two vertical columns, which form the answers and mapping sections of the table, are usually referred to as "rules".

Question	YES	NO
Action #1		X
Action #2	X	

Fig. 2-3 Simple table form.

Fig. 2-3 shows the form of a decision table which could describe a procedure consisting of one decision and two actions. There are two rules in the table, and this indicates that there are only two possible outcomes of the decision. The answer to the question will always be either yes or no. If the answer is yes then only action number two will be performed, while if the answer is no then only action number one will be performed.

When a procedure has been described in a decision table, the procedure can be carried out by executing the table. This involves the following steps: first, the question is evaluated; second, the answer to the question is used to select one of the rules, namely, the one with the matching answer; and third, the actions, if any, indicated on the selected rule, are performed. A specific example of a table which describes a very simple procedure appears in

Fig. 2-4. There is one question which will govern the

Is the temperature outside below 30°	YES	NO
Wear a light jacket		X
Wear a heavy overcoat	X	

Fig. 2-4 Table for choice of apparel.

choice of apparel. If the temperature is below thirty degrees, then wear a heavy overcoat; otherwise, wear a light jacket.

Many procedures involve more than one decision, and may be represented by listing all the questions and entering all combinations of the answers. Fig. 2-5 shows the form of a table which consists of two questions and three actions. When a table which contains more than one

Question #1	YES	YES	NO	NO
Question #2	YES	NO	YES	NO
Action #1	X		X	X
Action #2		X		
Action #3			X	X

Fig. 2-5 Table with redundancy.

question is executed, all the questions are considered to be evaluated in parallel. This evaluation results in the

selection of the one rule whose answers match the actual answers to the questions. A table is invalid if, for some combination of answers, it is possible to select more than one rule or no rule. In a valid table, the selected rule indicates which actions are to be performed. Since more than one action can be indicated by the entries on a rule, a convention is required to govern the order in which the actions are executed. Actions are usually executed sequentially from top-to-bottom of a rule.

In the table shown in Fig. 2-5, the third and fourth rules indicate the same set of actions. Thus, the same actions are to be executed whenever the answer to the first question is no, regardless of the answer to the second question. In other words, we don't care about the answer to the second question, under these circumstances. The table could be improved by rewriting it in the form shown in Fig. 2-6. The dash which appears in the third rule, is called a "don't-care". It indicates, in this example, that the second question is irrelevant when the answer to the first question is no. The use of don't-cares eliminates redundant information and therefore decreases the size of a table. More important is that the table is made less complex and easier to understand.

If don't-cares were not used, a table of N questions would contain two to the power N rules. With don't-cares there might be as few as N+1 rules. To

Question #1	YES	YES	NO
Question #2	YES	NO	-
Action #1	X		X
Action #2		X	
Action #3			X

Fig. 2-6 Table with don't-care.

illustrate the magnitude of this difference, consider a table which contains eight questions. Without don't-cares there would be 256 rules. The use of don't-cares could reduce the number of rules to as few as nine.

Some tables have rules which indicate the same set of actions, but which cannot immediately be combined by the introduction of a don't-care. In Fig. 2-7, rules one and

Question #1	YES	YES	NO	NO
Question #2	YES	NO	YES	NO
Action #1	X		X	X
Action #2		X		X

Fig. 2-7 Table with hidden redundancy.

three are candidates for combination. First, however, the table must be rewritten with the order of the two questions reversed, as in Fig. 2-8. Now rules one and two in the

Question #2	YES	YES	NO	NO
Question #1	YES	NO	YES	NO
Action #1	X	X		X
Action #2			X	X

Fig. 2-8 Table after reordering.

rewritten table can be combined by the introduction of a don't-care for question number one, when the answer to question number two is yes.

The definition of a don't-care implies the restriction that the first question in a table must not have any don't-cares among its answers. Also, if the answers for a question are coded entirely as don't-cares, then the question should be omitted since it does not participate in the selection of a rule.

Thus far, only "limited answer questions", those which can be answered with a response of yes or no, have been considered. (A don't-care is not an answer but rather an indication that the answer to a particular question is irrelevant.) As is illustrated in Fig. 2-9, tables can also contain "extended answer questions" which require specific answers other than yes or no. Such a table is interpreted in much the same way as one with only yes or no answers. In this example, the table prescribes that if the colour of the

What colour is the object	RED	GREEN	BLUE
Apply brown trim	X		
Apply yellow trim		X	
Apply orange trim			X

Fig. 2-9 Extended answer table.

object is red, then brown trim is applied, and so on. This table is not complete since red, green, and blue do not form an exhaustive list of colours and no provision is made in the table for an object of another colour. This is a potential problem which will be discussed in a later section.

It follows from the definition of a decision table that to enter a table at a particular question, and thereby ignore some preceding questions, cannot be allowed. Similarly, to enter at a specific action, or even rule, is not permitted. However, these misconceptions are frequently held by persons learning to use decision tables. To combat such errors, the teaching of decision tables must emphasize the concept that a table is a complete unit where the entire group of questions is evaluated so that a set of actions can be selected for execution.

2.2 History and Development

The key to the invention of decision tables was the recognition of the value of tables. Tables are widely used to record and communicate information in a highly intelligible, and often compact, form. Most people use an assortment of tables each day, and probably take them almost completely for granted. For example, a daily newspaper usually contains numerous tables such as a table of contents, tables showing prices for items and services being advertised, and tables of team standings for various sport leagues. Logicians have made use of truth tables for at least twenty centuries, according to London [3].

In 1957 or 1958, decision tables (or decision logic tables, or decision structure tables, as they have sometimes been called) were devised by a research group in the General Electric Corporation [4, pg. 3]. It is interesting to note that this team was not part of General Electric's computer hardware or software development groups but rather was involved in a study of a manufacturing process. They were searching for an alternative to narratives and flowcharts which they realized were seriously inadequate for recording the logic involved in a large and complex process. Very soon after formulating their new technique, these researchers recognized that computer programs could be prepared to check decision tables for

completeness, and detect errors such as ambiguous or redundant rules. This same group also proposed that computer programs might be written in a language employing decision tables to specify decisions, actions, and the relationships between them [5, 6]. They were suggesting a high-level language which would be translated into machine language by a compiler. This is particularly noteworthy since at that time high-level languages were just beginning to be accepted by the computing community. The first FORTRAN compiler had appeared only a couple of years earlier, the definition of ALGOL 60 was taking place at the same time [7, pg. 2], and the design of the COBOL language was not to begin until 1959 [8, pg. 1-2].

General Electric was also responsible for the next step in the development of decision tables. A number of organizations had investigated decision tables and begun to use them, at least as a manual technique, by 1961, when General Electric introduced TABSOL, a decision-table processor [9]. TABSOL was part of GECOM-II which was the primary programming language for business applications on their 200 Series computers. This was the first attempt by a computer manufacturer to encourage its customers to use a programming language based on decision tables and, in fact, it appears to have been the only such attempt. Unfortunately it is difficult to estimate the quantity of programming which actually made use of decision tables since a programmer

could code a complete program in GECOM-II, without using tables. Also, a substantial number of the 200 Series machines used a time-sharing operating system which was oriented to scientific users and did not support GECOM-II. Curiously, General Electric did not implement decision-table processors on its subsequent computer lines, perhaps because they recognized the trend towards using COBOL for commercial applications. This decision marks the end of the contribution of General Electric to the development of decision tables and to computer software for processing them.

The Conference on Data Systems Languages (CODASYL), which is responsible for the design, maintenance and publication of the COBOL language, is credited with the next advance in the development of decision tables. This was the realization that COBOL could provide a base for a decision-table programming language which could be largely machine independent and thus widely used. In 1962, a committee of CODASYL presented such a language, called DETAB-X. They proposed the concept of a preprocessor which would translate programs written in DETAB-X into complete COBOL programs. This idea spurred numerous organizations to experiment with decision tables, often by implementing a preprocessor for their own variation of the DETAB-X language. Many testimonials to the value of decision tables appeared in the literature during the following years. A book edited by McDaniel [10] contains a collection of such

articles in which, typically, the author describes the motives of his organization for using decision tables and the results. Unfortunately few authors describe in any detail the source language specifications for their preprocessor, or the standards for writing tables when they were used only in a manual system. Also lacking are articles chronicling unsuccessful attempts to use decision tables. It seems unreasonable to assume that there have never been failures, and an analysis of the factors which contributed to failure would be valuable to all who must avoid making the same mistakes.

The Special Interest Group on Programming Languages (SIGPLAN) of the Association for Computing Machinery (ACM) produced the next and to date last major step in the development of decision tables. This was the DETAB/65 preprocessor [11, pg. 146] which was implemented in COBOL, and translated programs written in a decision-table programming language into complete COBOL programs. The language was similar to DETAB-X but modified in several areas, primarily to make the language resemble COBOL as much as possible. The preprocessor was distributed freely and was successfully run on a number of different computers. The purpose of DETAB/65 was to encourage experimentation in programming with decision tables by making a preprocessor available to anyone whose computer had a COBOL compiler.

Since 1965, the main area of interest for research

on decision tables has been in developing algorithms for generating optimal object code from decision tables. The optimal code is that which minimizes both the amount of space occupied by the code and the time used to execute it. A substantial portion of the literature published on the subject of decision tables has been devoted to possible solutions to this problem [12, 13]. Several books have appeared recently which introduce the reader to decision tables and teach him to construct a procedure and write decision tables to express it [14, 15]. Such texts, however, often treat decision tables as a manual technique and almost ignore the concept of a computer programming language based on them. Almost completely missing from the literature is discussion of the factors involved in the design of a programming language which uses decision tables as the major control structure.

CHAPTER 3

Variations on Table Layout

In Chapter 2, the basic form and layout of decision tables were discussed, and a summary of the major events in their brief history was presented. The subject of this chapter is an examination of a number of features which have been suggested as useful extensions and modifications to the basic table layout.

3.1 Extended Answer Tables

It has been customary to make a distinction between three types of table. Those tables which consist entirely of limited answer questions are called "limited answer tables". Those containing only extended answer questions are called "extended answer tables", and a table containing a combination of both types of questions is a "mixed answer table". (Traditionally the word "entry" has been used instead of "answer" in these names in spite of the ambiguity of the term.) In order to shorten these names, the word "answer" will often be omitted. Fig. 2-9 is an example of an extended answer table, while Fig. 2-4 shows a limited answer table.

It is obvious that a limited answer table is a

special case of an extended answer table, in which all questions have exactly two possible answers, yes or no. It is possible to express any extended answer question in the form of one or more limited answer questions by writing a separate question for each of the extended answers. Fig. 3-1 shows a limited answer table which corresponds to the extended answer table in Fig. 2-9. A new action has been added to handle the case where a colour other than red, green or blue appears. The most undesirable feature of such

Is the colour red	YES	NO	NO	NO
Is the colour green	-	YES	NO	NO
Is the colour blue	-	-	YES	NO
Apply brown trim	X			
Apply yellow trim		X		
Apply orange trim			X	
Apply white trim				X

Fig. 3-1 Limited answer table.

a conversion is that the number of questions in the table usually increases, thereby making the answers section of the table more complex and less intelligible. Limited answer tables are sufficient to describe any procedure; however, in many cases it is preferable to be able to use extended answer tables.

The limited answer table has been predominant in the decision-table processors implemented to date, with the notable exception of TABSOL. It is interesting to see why extended answer tables have been omitted from most processors and even from some manual systems. The reason appears to be that extended answer tables are substantially more difficult to process because of their generality. Other complications arise in checking for completeness, ambiguity, and redundancy, and in the layout of the actual tables. These points will be considered in detail in later chapters.

3.2 Else Answer

The "else answer" (or "else rule" as it is often called) is a solution to the problem which was raised in connection with the example of an extended answer table in Fig. 2-9. The question in that table asked what colour an object was and listed the expected answers. No allowance was made for the case where the colour of the object was not one of those listed. The table is incomplete since it does not specify what actions, if any, are to be performed should this situation arise. The addition of an else answer, namely an answer consisting of "anything else", would complete the table by adding a rule on which to specify actions for the exceptional case.

Not all extended answer questions are incomplete;

for example, Fig. 3-2 shows a question where an else answer is not needed.

Is the value of A	< 0	= 0	> 0

Fig. 3-2 Complete extended answer question.

The limited answer table in Fig. 3-1 contains one more rule than did the corresponding extended answer table. This extra rule provides for the case where none of the three questions can be answered yes. Many authors suggest that such a table be written in the form shown in Fig. 3-3.

				ELSE
Is the colour red	YES			
Is the colour green		YES		
Is the colour blue			YES	
Apply brown trim	X			
Apply yellow trim		X		
Apply orange trim			X	
Apply white trim				X

Fig., 3-3 Table with ELSE answer.

Here an else answer is used to complete the table. In a larger and more complex table there might be a number of missing answers whose place would be taken by the else answer. Although such a practice is workable, it seems

unsatisfactory since it encourages one to write incomplete tables, and depend on the else answer to complete them. When using the else answer in this way, it is very easy to neglect to fully consider each of the omitted answers, and the actions which should be executed for each of them. This attitude can clearly lead one to write tables which are technically complete but are, in fact, incorrect representations of the process intended.

A reasonable conclusion would be to outlaw else answers in limited answer tables. This ban may result in the addition of several rules to some large limited answer tables in order to indicate exceptional cases which would otherwise have been expressed by the one else answer. In some cases, however, it will be possible to reorder the questions and eliminate some of these rules by the introduction of don't-cares. Such simplification improves the comprehensibility of the table, thereby making it preferable to the corresponding table with the else answer.

The same conclusion cannot apply when extended answer tables are considered. Despite the disadvantages of an else answer, some questions will require an else answer to be complete.

3.3 Ordering Within Rules

When a rule is selected for execution, the actions indicated on that rule are normally executed sequentially from top-to-bottom. It has been proposed [4, pg. 42] that a nonsequential order of execution of the actions on a rule might be specified. The ordering could be stated by replacing the X's in the mapping section of the table by the digits 1, 2, 3, etc. Then when a rule is selected, the action marked 1 would be executed first, 2 would be executed second, and so on.

Question	YES	NO
Action #1	X	
Action #2	X	X
Action #3		X

Fig. 3-4a Seq. actions.

Question	YES	NO
Action #1	1	2
Action #2	2	1

Fig. 3-4b Nonseq. actions.

The advantage of this strategy is that a shorter table may result. The top-to-bottom sequential ordering may force the dupliction of an action that is executed on more than one rule. Fig. 3-4a shows a table using sequential ordering. If actions one and three are identical, then the table can be rewritten as in Fig. 3-4b with nonsequential ordering and without repeating the action.

A nonsequential ordering suffers from a major

disadvantage in that it complicates the tables in which it is used. When one studies a rule in such a table it is necessary to continually search for the subsequent digit in order to locate the next action to be executed. This is certainly distracting and might be a real annoyance in a long rule where the ordering is quite irregular. Shortening a table by eliminating some actions is not nearly so valuable as eliminating questions or rules. Doing the latter is helpful since it simplifies the process of selecting a rule. Thus, the usefulness of specifying an ordering within rules is questionable, at best.

3.4 Compound Questions

Most instructional texts on decision tables imply that only simple questions may be used. By simple questions is meant those which test only one condition. Compound questions can be constructed, however, where two or more conditions are tested and are joined by the use of logical operators. The use of such a combination of tests can reduce the number of questions in the table and simplify the answers section.

One situation where the use of compound questions is advantageous is when a range test is performed on a variable. Fig. 3-5a shows a range test using only simple questions, while Fig. 3-5b shows the corresponding test

I ≤ 10	YES	YES	YES	NO
I ≤ 5	YES	YES	NO	-
I ≤ 0	YES	NO	-	-
High range			X	
Low range		X		
Out of range	X			X

Fig. 3-5a Simple questions.

I ≥ 6 AND I ≤ 10	YES	NO	NO
I ≥ 1 AND I ≤ 5	-	YES	NO
High range	X		
Low range		X	
Out of range			X

Fig. 3-5b Compound questions.

using compound questions. This illustrates an additional benefit gained from the switch to compound questions, namely that the end points of each range appear together in the same question. Thus, when compound questions are used, it is necessary to examine only one question to determine a complete range. However, if simple questions are used, more than one question, as well as the answers, must be studied to obtain the same information.

3.5 OR Tables

All the decision tables which have been discussed so far are AND tables, meaning that the questions in each table are connected by the logical operator AND. The OR table has been proposed, although not widely accepted, where the questions are connected by OR in some places as well as by AND in other parts of the table. Fig. 3-6 is an example

Quantity ordered ≥ Discount-quantity	YES	NO
Is the customer a wholesaler	YES	NO
Bill at discount rate	X	
Bill at regular rate		X

Fig. 3-6 OR table.

given by Gildersleeve [14, pg. 140] to illustrate the OR table. He uses the separation between rules to mark the switch from AND to OR. The meaning of this table is that if the quantity ordered is greater than or equal to the discount quantity AND the customer is a wholesaler then the billing is at the discount rate. If either the first question or the second or both are answered no, then bill at the regular rate. An OR table does effect a reduction over the corresponding AND table, in the number of rules required and therefore lessens the complexity of the answers section. This advantage is outweighed by the fact that the OR table

technique is extremely unwieldy in a table with a larger number of questions.

Fortunately there is a way to obtain the advantage of an OR table, namely, a reduction in the number of rules, while at the same time avoiding the negative aspect. The solution is to abandon the OR table and use, instead, compound questions in an AND table. The example in Fig. 3-6 can be rewritten with a single compound question formed by joining the two original questions by AND. Not only does this maintain the smaller number of rules but, in addition, it reduces the number of questions. In a large table where the OR table would be difficult to use, the AND table with compound questions is ideal since it reduces the number of questions, and simplifies the answers section of the table.

3.6 Horizontal Tables

In all the decision tables which have been examined above, the rules have been vertically oriented. It is interesting, however, to note that decision tables were first written with the rules stretching horizontally, and with the questions and actions written across the top of the table. Fig. 3-7 is an example of a horizontal table and demonstrates that exactly the same information is given in a horizontal table layout as would appear in the corresponding vertical table.

Question #1	Question #2	Action #1	Action #2
YES	YES	X	
YES	NO	X	X
NO	-		X

Fig. 3-7 Horizontal table.

If a horizontal table contains very many questions and actions, then there is only a very narrow area in which to write each question or action. Thus questions and actions must be abbreviated and broken over several lines, which makes them difficult to read. Alternatively, the writing must be turned and placed vertically, which is not feasible on either typewriters or line printers.

Horizontal tables are obviously of historical significance but do not appear to be suitable for practical use.

3.7 Extended Actions

The mapping section of a decision table usually contains X's which indicate the actions to be executed on each rule. Such actions are called "limited actions". In section 3.3 the possibility of specifying the order of execution for actions on each rule was studied. Another modification to the mapping section, which is worthy of

consideration, is to allow the use of "extended actions", where the action extends into the mapping rules. Instead of an X, part of the action would appear in each rule on which that action is to be executed. For example, if it is necessary to assign different values to a variable depending on the answer to a question, then the values to be assigned can appear in the rules. Fig. 3-8 shows such a case.

Is it leap year	YES	NO
Assign to DAYS the value	366	365

Fig. 3-8 Extended action table.

The use of this method for stating actions can shorten the actions section of a table. It eliminates all but one of a set of actions which are similar but have a corresponding variation, such as a value to be assigned. There is a disadvantage, however, in using extended actions, namely the visual disruption in the mapping rules. The pattern of X's is broken by mapping entries of varying size and formed from a variety of characters. Whether the advantage gained from shortening the actions section of a table outweighs the disadvantage incurred by making the mapping section more difficult to read is questionable.

Another consideration is that if it were decided to allow the user to specify the order of execution of

actions on a rule, and also use extended actions, then a sequence number as well as a portion of the action would have to appear in the rule. The two parameters could prove confusing with even the most imaginative delimiter to separate them.

3.8 Linkage

One can use decision tables for several purposes: as a tool for recording the logic involved in a procedure, and for communicating it to others; as the basis for a programming language; or as a technique for documentation. Regardless of one's purpose for using them, there must be some provision for linking tables together. Only a trivial process could be described by a single table. A number of points to be considered in designing this control structure will be discussed in the following sections.

3.8.1 Table Identification

A decision table should be identified by a unique name so that it can conveniently be referred to by commands appearing anywhere in the description of the same procedure. When decision tables are used as a manual technique, this name might appear as a title; in a programming language the name would conform to the rules for labels.

3.8.2 Table Initialization

It is often suggested that an additional section be included in decision tables. This new section might be called the "initialization" section, and would appear before the questions section. The content of this new section would be a group of actions to be executed before evaluation of the questions. The purpose of such a section is to allow the initialization of variables, opening of files, and initial "transput" (i.e., input/output) [16, pg. 126] operations before the questions in the table are evaluated.

There is a problem with including an initialization section in a table, namely, that it is sometimes necessary to be able to execute the table but skip the initialization section or at least part of it. It can be argued that there is little to be gained from making initialization actions a part of a table, although some way of including them in the description of a process is certainly required. An alternative to the initialization section is to allow actions which involve no decision making to be placed at any point outside a table. If these actions can have labels which are compatible with the labels identifying tables, then control statements can transfer either to a table, or to an action outside a table for initialization or reinitialization.

3.8.3 Open or Closed Tables

Decision tables are complete units in the sense that they include both a set of questions, and a group of actions to be executed depending on the answers to the questions. However, two types of table, "open" and "closed", can be formulated. In an open table, control falls through to the following table, or, if present, to the group of initialization actions preceding the next table. After execution of the last action in a closed table, control returns to the point where the evaluation of the table was requested. Thus an open table is analogous to a single executable statement, while a closed table resembles an internal subroutine. The choice between open and closed tables will lead to one of the two structures which are discussed in the following sections.

3.8.4 Flow Structure

The use of open tables allows sequential execution of a series of tables, one after the other, and perhaps including groups of initialization actions placed between the tables. However, this is not a flexible structure. At some point, depending on prior results, it may be necessary to select one of several paths, possibly joining together at a later point. If the paths are reasonably independent, then a separate decision table or tables could be coded for

each path, and a transfer mechanism used to switch control to one of these paths. A decision table could contain the questions involved in deciding which path to select, and actions (possibly one extended action) naming the tables which begin each path. This transfer action would strongly resemble the GOTO statement which is common to many programming languages including ALGOL, COBOL, FORTRAN and PL/I. When the end of each path is reached, this same transfer action can be used to return to the common path.

The simple transfer action can be used to skip execution of a table under some conditions, or to branch back and execute again a previous table or the table containing the transfer. Although this is a primitive looping facility, a more practical looping facility could be provided by a control statement similar to the FOR statement in ALGOL 60. It could be used to execute a table or group of tables repeatedly, or it might appear in the actions section of a table to cause the actions within its range to be executed a number of times.

3.8.5 Modular Structure

By using closed tables, it is possible to obtain quite a different structure than that proposed in section 3.8.4. Closed tables emphasize that each table is an independent unit. The control statement which facilitates

the use of closed tables is a transfer to a table with control returned when execution of the table is complete. This would be an internal transfer such that all variables available to the caller would also be available to the called table. This transfer statement resembles the PERFORM verb in COBOL and, in fact, could employ all the formats used for the PERFORM verb, thereby providing a flexible looping facility.

To use this modular structure to describe a procedure would involve writing two things: first, a set of closed decision tables; and second, a "driver" which would contain initialization actions and transfers to the closed tables to actually perform the processing. Since the driver would usually have to make decisions in order to determine what tables to execute, it would be advisable to write the driver in the form of decision tables.

The advantage of a modular structure is that it allows a separation of two different types of decisions: first, those decisions which control the paths through the procedure are concentrated in the driver; and second, the decisions which govern the actions required to carry out the procedure appear in the closed decision tables.

3.8.6 External Structure

Most programming languages provide a facility for executing subprograms, which are often compiled separately, and are not necessarily written in the same language. Typically, parameters may be passed between the routines, but both the calling routine and the called subroutine or function must conform to a regime of linkage conventions.

When decision tables are used as the basis for a programming language, it would be highly desirable to be able to access subprograms written in this same language or other languages. This would make it possible to encode and test a description of a procedure which could then be referred to from other procedures. Even when used strictly as a manual technique, a facility for the separate definition of commonly used procedures is valuable.

CHAPTER 4

Are Decision Tables Better?

4.1 Advantages and Disadvantages

Decision tables were invented about fifteen years ago by a research team which had become disenchanted with flowcharts and narratives. There were three areas in which decision tables were designed to be of value: first, as a tool for use in analyzing existing procedures and synthesizing new ones, to record the questions and actions involved and the logical connections between them, and to communicate this information to others; second, as the basis for programming languages; and third, as a technique for documenting programs.

In the following three sections the advantages and disadvantages of narratives, flowcharts, and decision tables will be discussed in light of the three purposes listed above.

4.1.1 Narratives

Narratives are probably the best of the three techniques for expressing the logic involved in very simple and small procedures. Narratives can be written and understood by almost anyone without special training. Similarly,

narratives are very well suited for the documentation of simple programs. Unfortunately, when a large procedure or program is involved, narratives are the least valuable of the techniques. This is because: first, they are far less amenable to standardization; second, they are much more likely to harbor incomplete and ambiguous logic; and third, they tend to be huge.

Narratives must be translated into a programming language before the process which they describe can be executed on a computer. This translation step provides a prime opportunity for errors to slip in. Since narratives are not directly processed by a computer, they require manual checking for ambiguity, redundancy, and completeness.

4.1.2 Flowcharts

Flowcharts have long been accepted as the only available tool which was practical for recording descriptions of procedures, and documenting computer programs. This is true despite the training which is required to write flowcharts, or even to read them, and two other major drawbacks, namely, the graphics element involved, and the tendency to discourage modularity.

To a large extent, flowcharts are not written, but rather are drawn. Much of the effort required to produce a good flowchart is devoted to preventing the interconnecting

lines from degenerating into an indecipherable maze. Often the wording on a flowchart has the lowest priority, and is abbreviated and squashed to fit into a standard sized symbol.

Flowcharts tend to be written as continuous streams of decisions and actions, not broken into natural segments of the procedure, but rather, chopped arbitrarily to fit on pages.

Primarily because of the graphics element involved, flowcharts are not a practical basis for a programming language. To enter a flowchart into a computer would probably require an interactive graphics device and considerable software. Computer storage of a flowchart such that it could be regenerated in the form in which it was entered would require a complex data structure, and would probably prove very expensive to process. Like narratives, flowcharts are translated into a programming language before entry to a computer. This extra step is time consuming and allows errors to creep into the description of the procedure. Since the computer does not actually process the flowchart itself, it cannot check the chart for errors or for adherence to standards. If any checks are made they must be done manually, and preferably not by the person who produced the flowchart, but rather by someone more objective.

4.1.3 Decision Tables

Decision tables are a relatively new technique, and suffer from the disadvantages that they are not yet widely accepted or used. Like flowcharts, they require training and practice before one can begin to use them efficiently. However, there are numerous advantages gained by using decision tables, that make them vastly superior to either flowcharts or narratives.

Decision tables are well suited for use as a tool for recording a description of a procedure, and also for use as documentation for computer programs. The most important feature of decision tables is that they can form the basis for a powerful programming language. Thus the tables produced by an analyst can be compiled into machine instructions without an intermediate manual translation into a programming language. It is not claimed that decision tables alone form complete programs, since a program usually must contain some nonexecutable statements such as declarations, as well as initialization actions (which are discussed in section 3.8.2). If a program uses decision tables to specify the logical structure of a procedure being implemented, then the program may be largely self-documenting. The decision tables which form the documentation are, in fact, the major part of the source program. This point is of great importance since any change in the program

automatically means that the documentation is updated at the same time. When either flowcharts or narratives are used as program documentation, a modification to the program necessitates a corresponding but separate change in the documentation. Since programmers typically look upon changing a flowchart or narrative as a dull and uninspiring task, it requires strong discipline to avoid a situation where programs and documentation do not agree.

Decision tables are compact and do not require the graphical component of flowcharts. A table showing several questions, a number of actions, and a complex mapping between them, can be written in less than a page. A flowchart for the same procedure might use several pages and require substantially more time to draw. The result would probably be a large flowchart which would be far more difficult to understand than the compact decision table representing the same procedure. This compactness is also apparent in programming where one programmer must juggle both a source listing of the program and a large flowchart while another works with a single listing of the decision tables and the initialization statements which together form his program.

It was noted above that decision tables force the user into a modular style while flowcharts discourage modularity and encourage one to describe a procedure as a long stream of questions and actions with a maze of

interconnections.

When decision tables are used as a programming technique it is feasible to use the computer to check each table for completeness and to ensure that there are no ambiguities or redundancies. The effect of such testing should be to detect errors at the compilation stage. Without these checks such errors might remain hidden until the testing stage or even until the program is run on live data, and is being depended on to produce correct results.

4.2 Why Doesn't Everyone Use Decision Tables?

Despite the conclusion that decision tables are superior to both flowcharts and narratives, there has been no universal acceptance of decision tables. A variety of factors which contribute to this lack of acceptance will be discussed in the following paragraphs.

The primary reason for the slow growth in the use of decision tables is that flowcharts have been in use longer and have themselves obtained virtually universal acceptance. This leaves decision tables in the position of having to unseat the incumbent. To do so it is necessary to convince the policy makers, in the numerous organizations which use flowcharts, of the superiority of decision tables, and to overcome their resistance to change, and finally to actually train analysts and programmers to use the new

technique. Since there is no major organization, such as a computer manufacturer, actively pushing decision tables, the slow growth in their use will probably continue.

Many organizations which do try to use decision tables restrict themselves by using them only as a documentation tool. Since this is only one of three areas where decision tables are valuable, many of their potential benefits are not realized. It is not difficult to understand why an organization, especially a small one, might be reluctant to begin to use decision tables as a programming technique. To do so would require the design, implementation, and maintenance of some new system software. The distribution of DETAB/65 was an attempt by SIGPLAN to foster the use of decision tables by reducing the difficulty and cost of installing an operational preprocessor. Another objection to the use of decision tables for programming is that preprocessors such as DETAB/65 introduce a new level of overhead, namely the computer resources used to translate the decision-table language program into a language such as COBOL. Both of these objections can be countered by noting that one must make an investment before one can expect a gain. The costs involved in installing a preprocessor and the extra costs of using it should be repaid many times over through the greater productivity of the programmers and analysts using decision tables. This is the same argument which justified the switch from the use of assembler

language to higher-level languages for the great majority of applications programs.

Almost everyone who takes an introductory computing science or programming course, either at an educational institute or from a computer manufacturer, is taught flowcharting. When a person learns flowcharting, he is learning a technique which will influence his programming style. Unfortunately, flowcharting appears to encourage a highly scattered program organization where almost every decision results in at least one branch. A language such as FORTRAN fits this style perfectly due to its very rudimentary control structures. In fact, when programming in FORTRAN, this scattered organization is very difficult to avoid. This is unfortunate because a scattered organization makes it extremely difficult to understand and follow the logic of the procedure, even with a corresponding flowchart. Decision tables, however, have quite a different structure. They tend to force modularization since related questions and actions are grouped together into decision tables which appear as single units. The radical difference between the two techniques probably accounts for a large part of the resistance to decision tables. Even a person who is well disposed to learning the new technique, decision tables, will require some time to adapt to it and gain experience with its use. During this period of time he may be discouraged if he feels he is working with a lower degree of

efficiency than he could achieve using his older, well practiced technique of flowcharting. Such feelings may even convince him to discard the new technique in spite of the long term benefits he would gain by mastering it.

One other factor which may alienate a programmer from decision tables is the awkwardness of keypunching a table, or making modifications to an existing table. To illustrate this problem, which stems from the rigid layout of a table, consider what is involved in adding a single question to a table which has already been keypunched. It is likely that most of the cards which form the table will have to be repunched since one or more rules must be inserted. If the programmer has to punch his own changes, this could prove to be time consuming and annoying. Fortunately, this problem can easily be overcome by using the computer to make these tasks more convenient and pleasant. If the operating system in use allows conversational access to the machine, then only a relatively small amount of additional software may be required. One approach might be to extend the system's context editor [17, pg. 289] to provide commands designed to facilitate manipulation of tables. If a light-pen equipped CRT is available, then the light pen might be a convenient pointer with which to indicate where to insert or delete rules or rule entries. The availability of such a facility contributes to building a "good" working environment. Although it is difficult to

measure precisely, it is generally agreed that one's environment significantly influences one's productivity [18, pg. 191].

It seems reasonable to conclude that decision tables will become universally accepted and used, only if people are taught to appreciate their advantages and use them when they first learn the concepts of programming. As long as only flowcharting is taught at this crucial time there will always be resistance and difficulty in switching at a later time.

It is not suggested that either flowcharts or narratives be completely abandoned. Flowcharts appear to be useful where a graphical element is of value and there is little decision making, for example, in a general description of a system of programs showing the input and output files for each program. Narratives are a practical technique for describing small procedures which do not involve any complex decision making.

CHAPTER 5

Language Design

In the previous three chapters, the reader has been introduced to decision tables. In this chapter, a decision-table language will be proposed, and the problems encountered in designing the language will be discussed. The implementation of a processor for this language will be the subject of the next two chapters.

5.1 Design Approach

There are two quite different approaches to designing a decision-table programming language: first, design a completely new language; and second, use an existing language as a base on which to add decision tables, as a new control structure. Unfortunately there are disadvantages with each approach.

The primary disadvantage of designing an entirely new programming language is the very large amount of work which is required. Even worse is the fact that much of this work is quite unrelated to decision tables. For example, a new language usually requires definition of basic concepts such as character set, modes, scope rules, and the form of identifiers, numbers, strings, and reserved words. Also

needed are facilities for making declarations, rules for forming expressions, and provision for transput operations. All these are in addition to the control structures, where decision tables are the focal point.

The alternative to designing a new language is to use an already existing programming language which has suitable basic concepts and facilities, and which, in addition, has a control structure that can be modified or replaced. This approach can result in a substantial saving in time and effort. The main disadvantage is that since the language was not designed with decision tables in mind, some undesirable features or awkward restrictions may be imbedded in the language. There are, however, a number of advantages to be gained with this approach. Documentation and instructional materials for the existing language can probably be quickly adapted for use with the decision-table language. If the users are already familiar with the existing or "host" language, then teaching the extended language may be reduced to little more than instruction in the use of decision tables. Similarly, resistance to using decision tables for programming may be reduced by introducing them in familiar surroundings, that is, in a language which the people involved have already accepted and have experience with.

In this study of decision-table programming language design and implementation, the second approach will be

employed. Thus the discussion can be centered on decision tables rather than on the much broader subject of programming language design in general. FORTRAN was chosen as the existing language to be modified and extended. This choice was prompted both by the widespread use and acceptance of FORTRAN, and by the substantial improvement which is made to the language, by the introduction of decision tables as a new control structure.

Since FORTRAN does not include a block structure, in the ALGOL 60 or PL/I sense, and because of the limited forms of IF statements which are available, the user is forced to code a large number of otherwise unnecessary transfers. These frequent transfers make it difficult for anyone to follow and understand the logic involved in a program [1, 2]. When decision tables can be coded, IF statements need no longer be used, and instead, related questions and actions are coded together in a structure that is very easy to understand. FORTRAN is not usually considered to be a language which encourages a highly modular style of programming. However, when decision tables are used, it is difficult to avoid such a style.

Having chosen to extend an existing language, FORTRAN, a secondary design goal is apparent. This is to make as much use as is possible of FORTRAN conventions, and to minimize the number of new conventions and restrictions.

5.2 Table Components

The first stage in designing the syntax for writing a decision table in a programming language, is to study the requirements for each of the four components: questions, actions, answers, and mapping. This will be done in the following four sections.

The requirements for both limited and extended answer questions and actions will be considered. In a later section of this chapter, tables consisting entirely of limited answer questions and limited actions, will be treated as a special case in which a simplified syntax is possible and advantageous.

5.2.1 Questions

In the previous chapters, several sample decision tables have been shown. Two types of questions were asked in those tables: in one type, the answers were always either yes or no; and in the other type, the expected answers were specified by the person writing the table. The two were referred to as limited and extended answer questions, respectively.

The limited answer question is analogous to a FORTRAN logical expression, which can be evaluated to produce a value of .TRUE. or .FALSE., corresponding to the yes or no answers. A logical expression can consist of any

combination of the following three components, all joined by logical operators: first, arithmetic expressions joined with relational operators; second, logical variables; and third, logical function references [19, pg. 24]. Thus the syntax required for stating simple and compound limited answer questions is exactly the FORTRAN syntax for logical expressions. For example, "I .LE. 10" and "I.GE.6.AND.I.LE.10" are logical expressions, and thus valid questions. They correspond to the first question in Fig. 3-5a and Fig. 3-5b, respectively.

Extended answer questions cannot be encoded simply as FORTRAN logical expressions, since, unlike limited answer questions, they do not have a binary, yes or no, type of answer. A typical extended answer question is: "What is the value of Z?" The possible alternatives, namely the possible values of Z, are listed in the answers. These values might, for instance, be 1, 2, or 3. The question can be rewritten in the form, "Is Z equal to 1, to 2, or to 3?". For the moment, the case where Z is not equal to any of the listed values, will be ignored. The else answer, which was introduced in section 3.2, handles this case. It is discussed in detail in section 5.2.3. A further rewriting of the above question is "Z .EQ. {1 or 2 or 3}". Another example of an extended answer question written in this form is "A {.LT. or .EQ. or .GT.} 0" which corresponds to the question in Fig. 3-2.

These last representations of extended answer questions suggest a form for encoding such questions. The lists of alternatives which appear in braces in the above examples, contain the answers and can be coded apart from the remainder of the question. Section 5.2.3 deals with the answers section and proposes a syntax for coding the answers to both limited and extended answer questions. The question can be written as a logical expression with a missing term. The word "term" is used to refer to what has been omitted, namely the extended answers. This missing term would usually be a constant or a relational operator, as in the two examples above, or an arithmetic expression.

The location of the missing term within the logical expression can be indicated or determined in several ways: first, by the use of a fixed location; second, by the use of a symbol to mark the location; and third, by other forms of syntactic analysis of the logical expression.

The fixed-location method is the simplest but it is also the least flexible. The last term of the logical expression would probably be the most convenient to pick as the fixed term, but then the second example above could not be encoded.

Syntactic analysis appears to be a practical alternative, but there are two problems. One problem is that a much more restrictive definition of a term would be required. For example, consider the question "CODE .EQ.

VAL{A or B or C}" where the braces contain three possible final characters for the identifier which begins with "VAL". The question would appear alone as "CODE .EQ. VAL", with the answers coded separately. There are nine syntactically correct positions in this question from which the answers could have been omitted. Obviously a convention would be required to indicate what can and cannot be omitted. A more serious problem is that this method tends to hide the location of the missing term. Anyone who reads the question must make an effort to figure out what is missing.

The second method, using a symbol to mark the location of the missing term, appears superior. It is flexible since the missing term can be located anywhere in the logical expression, and since the location is explicitly indicated. The only restriction involved is that the symbol used as the marker must be one which would not otherwise appear in a logical expression. The underscore character, "_", is proposed for this purpose since it implies that something is missing and must be filled in. Using this method, the three questions used as examples above would appear as, "Z .EQ. _", "A _ 0", and "CODE .EQ. VAL_".

Two extensions to this method of encoding extended answer questions are possible. The first would facilitate the use of questions such as "Are both A and B equal to 1, to 2, or to 3?". This question would be coded as "A.EQ._.AND.B.EQ._". This use of more than one underscore

indicates that the same value has been omitted from more than one place in the question. The second extension would be to allow omission from a question of two or more different terms. An example of a situation where this would be useful is for a range test on a variable. There the extended answers would consist of several pairs of values. This would require the definition of another delimiter symbol to separate the different values forming each answer. If these two extensions were not mutually exclusive, then a convention would be required to show which value in an answer corresponds to which missing term in the question. This complication is sufficient reason to rule out the combination of these two extensions.

Thus, all questions, whether limited or extended answer, are coded as logical expressions. Complete expressions are used for limited answer questions, and expressions with a missing term are used to represent extended answer questions.

5.2.2 Actions

The actions section of a decision table contains a set of actions. The actions are considered to be independent, and all, some, or none of them may be selected for execution for each combination of answers to the questions in that table. Just as there are two types of questions,

there are two forms of actions, also named limited and extended.

The syntactic requirements for limited actions are straightforward. Any FORTRAN executable statement can be an action. The FORTRAN rules for forming assignment, control, and transput statements are satisfactory and no restrictions are needed.

An extended action is one in which part of the action is selected from a list of alternatives written in the mapping section of the table. This, of course, is analogous to an extended answer question where a list of possible answers is written in the answers section. In order not to introduce restrictive rules and thus cause a loss of generality, it is proposed that any FORTRAN executable statement may be coded as an extended action. Similarly, the programmer should be free to select any part of a statement to be placed in the mapping section.

For example, control statements such as GOTO and DO may be extended as "GOTO {10 or 20 or 30}" and "DO 40 I={1,10 or 2,20,2}". A number of components of transput statements could be placed in the mapping section: the logical unit; FORMAT statement number; list of variables to be transmitted, or part of the list; and even the words READ or WRITE. Assignment statements are also well suited for use as extended actions; any part of the logical or arithmetic expression, or the variable to which the result

is to be assigned, could appear in the mapping section.

The location of the missing portion of the action must be indicated in some way. Marking the location with a symbol, such as an underscore, which was the method proposed for use with extended answer questions, appears to be the most practical procedure in this case as well. It is flexible, since any part of an action can be placed in the mapping section, and at the same time, the location of the omitted segment of the action is clearly marked. Using this method, the questions in the examples above would be coded as "GOTO _" and "DO 40 I=_".

Thus, any FORTRAN executable statement can be used as an action. Complete statements are used for limited actions, and extended actions are written with a missing portion which is to be filled in from the entries in the mapping section of the table.

5.2.3 Answers

For each question in a decision table, there is a corresponding set of answers in the answers section of the table. Although there is a difference in the _form of the answers, depending on whether a limited or extended answer question is involved, one syntax will be proposed for coding either type of answers.

The answers for a limited answer question are

either yes or no. These answers may be coded by using the words YES and NO, or alternatively, TRUE and FALSE. Abbreviations such as Y and N, or T and F, may be convenient and are acceptable since they do not significantly camouflage the intended answers.

The answers to an extended answer question do not come from a small set of valid answers as was the case with limited answer questions. Instead, each extended answer may contain one or more characters and consist of almost any combination of letters, digits, and some special characters. There should be no restrictions on the use of imbedded blanks.

At least two answers are required for any question, and are usually written side by side. This indicates that there must be some way to determine where one answer ends and the next begins. There are several ways to do this: first, use a fixed width for all answers; second, require the programmer to specify either a single width for all answers, or individual widths for each answer; and third, mark the division between answers with a symbol.

The first method, using a fixed width for all answers, is very inflexible and thus very wasteful. The width would have to be sufficiently large to accommodate the largest answer, but this could be far more than is needed for a small answer. Trying to establish the length of the largest answer might prove rather frustrating as well.

It is somewhat more practical to have the programmer specify the width. The benefit to be gained from this method is directly related to the level at which the width is specified. One width for all answers in a table is much less useful than stating a width for all the answers on each rule or for each question. Ideally, a width could be given for each answer of each question. Unfortunately, specifying more than one width per table could prove to be time consuming and rather tedious for the programmer.

The use of a delimiting character to indicate the end of one answer, and the start of the next, seems to be the most flexible and practical method. It is a convenient way to indicate the width of each answer individually, at the cost of adding one extra character to each answer. The symbol chosen as the delimiter must be one which would not otherwise appear in an answer. A blank would enhance the readability of the answers; however, its use must be ruled out unless imbedded blanks are excluded from the answers. The vertical bar, "|", is proposed for use as the delimiter since it has sometimes been used to indicate OR, and thus suggests a choice between the possible answers.

There are two other features of decision tables which further complicate the syntax required for the answers, namely, don't-cares and else answers.

A don't-care indicates that the answer to a question is irrelevant on a particular rule, that is, for a

particular combination of answers to the other questions in the table. The minus sign, "-", has been used in examples in the preceding chapters to indicate don't-care. Unfortunately it is not a suitable symbol for use in the answers, since an extended answer might contain part of an arithmetic expression including a minus sign. In fact, if only an arithmetic operator is omitted from an extended answer question, then an answer may consist of a minus sign. Clearly, some other character is needed if special rules or restrictions are to be avoided. The blank is a suitable character to indicate don't-care, and can be coded as one or more blanks between delimiters. Since a don't-care may appear as the first or last answer for a given question, it is necessary to mark the start of the first answer and the end of the last. The delimiter symbol which is used to separate answers, can fill this role as well.

The else answer which was discussed in section 3.2, is a way of entering "anything else" in the answers section. It handles those cases where the answer to an extended answer question is not one of the answers explicitly listed. The else answer can appear in two forms, "local" and "global".

The difference between local and global else rules, and the need for both forms, can be seen only in tables containing more than one question. Fig. 5-1 shows the questions and answers sections of a table consisting of

Is the value of A	0	0	0	1	1	1
Is the value of Z	1	2	3	1	2	3

Fig. 5-1 No else answer.

two extended answer questions and no else answer. Fig. 5-2 contains a slightly modified version of that table. The

Is the value of A	0	0	0	1	1	1	
Is the value of Z	1	2	3	1	2	3	

Fig. 5-2 Global else answer.

modification is the addition of a global else answer which appears as the rightmost rule. In this example, its meaning is that if the value of A is other than zero or one, or the value of Z is not one, two, or three, then the else rule is selected. The else rule makes it possible to specify a set of actions to be executed whenever either variable is not equal to one of the stated values.

Fig. 5-3 contains another version of the sample table. In this version, the global else answer has been replaced by three local else answers, each of which is indicated by a question mark, and has the meaning "any other value". The purpose of the local else answers is to permit different actions to be executed for each case in which the answer to a question is not one of the explicitly stated set

Is the value of A	0	0	0	0	1	1	1	1	?
Is the value of Z	1	2	3	?	1	2	3	?	-

Fig. 5-3 Local else answers.

of answers. In this third table, the actions indicated on rule nine will be executed whenever the value of A is not zero or one. Rules four or eight will be selected when the value of Z is not one, two, or three, and A is zero or one respectively.

Both global and local else answers can appear in the same table, as is illustrated by the table in Fig. 5-4.

Is the value of A	0	0	0	1	1	1	?	
Is the value of Z	1	2	3	1	2	3	-	

Fig. 5-4 Global and local else answers.

The table in Fig. 5-3 has been modified by the deletion of rules four and eight, and the addition of a global else answer which again appears as the rightmost rule. When both local and global else answers appear in a table, the local else answer has priority. Thus, in this example, rule seven will be selected when the value of A is neither zero nor one. The global else rule is selected when A is either zero or one, but Z is not one, two, or three. The global else rule is selected because local else answers were not

included in the table for these two cases.

Fig. 5-5 illustrates that a local else answer can be used in a table as though it was a specific answer to a question. In this example, when the value of A is not zero

Is the value of A	0	1	?	?	?	?	?
Is the value of M	-	-	4	5	?	?	?
Is the value of Z	-	-	-	-	8	9	?

Fig. 5-5 Hierarchy of local else answers.

or one, the selection of a rule is dependent on the values of M and Z. There are five rules, and thus five different sets of actions, from which to choose when the value of A is other than zero or one.

The syntactic requirements for the two types of else answer are quite different. A local else answer, which can be included among the answers to any extended question, should be encoded as though it were merely another possible answer. It must also be recognizable as an else answer and not be mistaken for an explicitly stated action. A symbol which cannot otherwise appear in an answer and which is different from the delimiter symbol, would serve this purpose. The question mark, "?", is proposed since it suggests doubt, and the local else answer indicates doubt as to other possible answers for a particular question.

There are several ways of encoding a global else

answer. One technique would be to enter the word "ELSE" as an answer. If it were entered on the first set of answers in a table, then no entries would be required below it in the same rule. Unfortunately this would create a reserved word, since it would not be possible to distinguish it from the identifier "ELSE". This problem could be avoided by inserting a symbol, such as "?", to form for example, "?ELSE". Clearly a shorter word, or perhaps just a symbol, would be preferable. It is proposed that a global else answer be indicated by a blank, in fact, a rule consisting entirely of blanks. Such a rule would not be confused with a rule containing some don't-cares, since a don't-care cannot appear among the answers to the first question of a table.

If an acceptable encoding scheme could be found, then an intermediate else answer might be useful. Its priority would be between the local and global else answers. It would appear as a global else for a subset of the answers in a table.

5.2.4 Mapping

The mapping section of a decision table indicates which actions, if any, are to be executed for each combination of answers to the questions in that table. A row of mapping entries accompanies each action, and contains one

entry for each rule, that is, for each combination of answers. The form of each entry depends on whether a limited or extended action is being mapped.

A limited action is a complete FORTRAN statement and does not extend into the mapping section. Thus, a limited action mapping entry need only indicate "execute" or "do not execute" this action. In this situation, the symbol "X" has traditionally been used to indicate the affirmative; and blank, the negative. This use of a blank does not create any problems; however, "X" is a questionable choice for the opposite role since "X" is also a frequently used variable name in FORTRAN programs. Since there is no other symbol with any mnemonic value, it is proposed that "X" be used for this purpose.

When an extended action is to be executed, part of the action appears in the corresponding mapping entry. When the action is not to be executed on a particular rule, blanks are placed in the mapping entry. In much the same way as for answers, the width of the mapping entry must be either fixed, specified by the programmer, or indicated by the presence of a delimiter. The use of a delimiter character is again the best choice, for the same reasons as stated in section 5.2.3. Since a mapping entry may consist of blanks, the start of the first entry and the end of the last, must be indicated as well. The symbol "|", which was proposed for both these purposes in the answers section, is

also appropriate for similar use in the mapping section. This is very fortunate since the layout of both the answers and mapping sections is then consistent.

5.3 Joining the Components

In the preceding sections, the syntax required for each of the four components of a decision table was discussed. A program written in FORTRAN is composed of a series of statements. Since the decision-table programming language is to follow FORTRAN conventions, a decision table will also consist of statements. The problems involved in joining the components to form statements will now be considered.

5.3.1 QUESTION/ANSWER Statement

Each QUESTION/ANSWER statement consists of one question followed by its corresponding answers. Either a limited or extended answer question may be used. The statement is coded in columns 7 to 72, and uses the syntax for writing the question and answers which has already been proposed. Since the symbol "|" cannot appear in a logical expression, the first occurence of a "|" in the statement indicates the end of the question and the start of the answers.

In FORTRAN, statement numbers are defined on those

statements which are referred to by control statements. The questions in a decision table are treated as a group, and are never transferred to individually. Since there is no need to be able to code a statement number on a question, columns one to five of all QUESTION/ANSWER statements should be blank.

It is desirable to define a continuation convention since a single source record may not always be long enough to contain a complete question and all its answers. The FORTRAN practice of entering a character other than blank or zero, in column six of each continuation card (record), is satisfactory for the QUESTION/ANSWER statement. However, some indication is needed of what is being continued: either the question, the answers, or both. One solution would be to insist that all the answers to a single question appear in the first source record and that only the question be continuable. This is reasonable since it would be very difficult to read and understand a table in which the answers for even one question are split over two or more lines. Fortunately, the problem may be overcome by taking advantage of the fact that a line printer, which is the device normally used to produce a program listing, typically has a print line of at least 120 characters. Thus even if the answers for a question are continued over several source records, it may be possible to print them on one line. Since this problem can be overcome, it is useful to permit

continuation of answers as well as of questions.

The method for indicating the end of the question, and the start of the answers, on the first source record, can also be used to separate the question from the answers on continuation records. The only restriction is that an individual answer must not be continued. Instead, the break should occur at the delimiter separating any two answers. This restriction eliminates the need for a convention to distinguish between a continued answer and two separate answers. This continuation technique still allows a flexible statement layout. When a QUESTION/ANSWER statement is continued, any of the records forming the statement can contain all or part of the question. An example of a QUESTION/ANSWER statement is the following:

```
I .GE. 6 .AND. I .LE. 10     | YES | YES | NO | NO |
```

The following three statements contain the same question and answers as the preceding statement, but serve to illustrate some of the layouts which can be obtained using continuation. Asterisks are used to indicate continuation in these examples:

```
      I .GE. 6
*       .AND.       | YES | YES | NO | NO |
*     I .LE. 10
```

```
      I .GE. 6 .AND. I .LE. 10      | YES | YES |
*                                   | NO  | NO  |


          I .GE.  6     | YES | YES |
*           .AND.
*         I .LE. 10                 |  NO |  NO |
```

5.3.2 ACTION/MAPPING Statement

Each ACTION/MAPPING statement consists of one action, either limited or extended, followed by its corresponding mapping entries. The statement is written in columns 7 to 72 of a source record, and uses the syntax for actions and mapping which was proposed above. Since the symbol "|" cannot appear in any executable statement, the end of the action and the start of the mapping entries are indicated by the first occurrence of a "|" in the statement.

Although an explicit transfer to an action in a decision table is not permitted, statement numbers may still be required on some ACTION/MAPPING statements. The DO statement specifies a statement number to indicate its scope. Thus, if DO is to be a valid action in a decision table, the final action of a DO loop must have a statement number. Otherwise a new convention would be required to indicate the scope of a DO. Even if a reasonable convention could be invented, the objective of using FORTRAN conventions whenever possible would not be met.

A continuation convention is required for ACTION/MAPPING statements, so that either the action, or the mapping entries, or both, may be continued onto successive source records. The convention which was described in the preceding section for use with QUESTION/ANSWER statements, is applicable in this case as well, and allows a flexible statement layout.

5.4 Forming a Complete Table

This section investigates what facilities are required, in addition to QUESTION/ANSWER and ACTION/MAPPING statements, in order to form a complete table.

The first point to be considered is whether or not to mark the start of a decision table in any special way. Because a QUESTION/ANSWER statement can be distinguished from any FORTRAN statement, an explicit signal is not absolutely necessary. Nevertheless, an explicit signal should assist the reader of a program to locate the start of a table. Two possible forms for this signal are: first, columns one to five of the first QUESTION/ANSWER statement could contain some word or symbols distinguishable from a statement number, to indicate the start of a table; and second, a new statement could be invented for this purpose.

A serious objection to the first method is that many tables will require the definition of a statement

number so that the table can be referred to in control statements appearing in the program. The obvious place for this statement number is the first statement of the table, but this would not be possible if columns one to five of the first statement were used for the start-of-table mark.

The more practical alternative is the second method, which is to invent a new statement, perhaps called the "TABLE" statement. It would consist of TABLE, entered in columns 7 to 72 of a source record. The convention which permits imbedded blanks to appear in any FORTRAN statement can be extended to this statement as well. A statement number could be entered in columns one to five so that the table could be referred to in control statements.

Thus a decision table will begin with a TABLE statement, followed by one or more QUESTION/ANSWER statements. Next will be one or more ACTION/MAPPING statements; however, it is proposed that an explicit signal be used to separate the top and bottom sections of a table. Another new statement, perhaps called the "ACTIONS" statement, could serve this purpose. The statement would consist of ACTIONS, entered in columns 7 to 72 of a source record. Since a transfer to this statement would not be valid, columns one to five should be blank.

It is proposed that the end of a decision table, like the start and middle, be marked with an explicit statement. There is another purpose for this statement,

namely, to indicate what is to be done after execution of the table. In an open table there are two possibilities; either the statement immediately following the table is to be executed next, or a transfer statement which was coded as an action causes a jump to some other statement. In the latter case, mapping entries indicate those rules in which a transfer action is selected.

It is proposed that the statement to mark the end of the table be coded as an ACTION/MAPPING statement, with "EXIT" as the action. In an open table, the meaning of this action will be that the statement following the table is the next to be executed. Thus any rule which does not include selection of an action causing a transfer out of the table should contain an "X" on this final action. A different meaning will have to be attached to EXIT in a closed table. In this case it indicates a return to the point at which execution of the closed table was initiated.

The FORTRAN CONTINUE statement is also a candidate for the role of marking the end of a table. It would be a poor choice since it will also be used in tables as the final action of a DO-loop. To give CONTINUE both meanings might prove rather confusing.

Failure to select the EXIT action would not necessarily cause any ambiguity, since the absence of a transfer implies that the statement following the table is to be executed next. Nevertheless, requiring selection of

either a transfer or the EXIT is a convenient way to force the programmer to consider, for each rule, what is to be done after the actions have been executed. Given this requirement, it will be possible to detect tables in which the programmer has forgotten to indicate what is to be done.

```
    T A B L E
A .EQ. _          | 0| 0| 0| 1| 1| 1|  |
Z .EQ. _          | 1| 2| 3| 1| 2| 3|  |
  A C T I O N S
I=I+_             | 2| 3| 9| 6| 4| 1|  |
Q=125             | X|  | X|  |  |  |  |
R=87              |  |  |  | X|  | X|  |
GOTO _            |  |60|  |  |70|  |  |
WRITE(6,50)A,Z    |  |  |  |  |  |  | X|
STOP              |  |  |  |  |  |  | X|
    E X I T       | X|  | X| X|  | X|  |
```

Fig. 5-6 A complete table.

An example of a complete decision table, using the five statements which have been described above, is shown in Fig. 5-6.

Among the actions in a table, it is reasonable to expect to find some transput statements, such as the WRITE statement in the table in Fig. 5-6. FORTRAN transput statements usually specify a FORMAT statement, and some programmers prefer to code a FORMAT immediately following the transput statement which specifies it [20]. For this reason, it is desirable to allow FORMAT statements to appear in a table among the ACTION/MAPPING statements. Since it is not an action, no mapping entries can be coded. A FORMAT statement is then the second situation (the final statement

of a DO-loop is the first) where a statement can appear inside a table.

The role of comment statements is to document a program. Comments are typically used in two ways: first, to define the general purpose of large groups of statements; and second, to explain the specific effects of individual statements or small groups. One of the most important benefits gained by using a decision-table programming language, is that programs tend to be self-documenting. This means that the need for comments is substantially reduced. Nevertheless, some comments are likely to be of value in almost any program, and should be permitted even inside decision tables. Comments inside tables could be used to clarify any obscure or confusing aspect of a question or action. The FORTRAN convention of entering a "C" in column one of a record to indicate that it is a comment, is acceptable for use with comments appearing inside tables.

5.5 A Complete Program

In the preceding sections of this chapter, a syntax for decision tables has been developed. What remains to be considered is the form of a complete program. The only change to the FORTRAN definition of a program that is necessary to accommodate decision tables, is to consider each table to be an executable statement. It may be

desirable to delete the two FORTRAN IF statements from the decision-table language since their function can be performed by decision tables. No restrictions are necessary on any features of the FORTRAN language.

In order to provide a way of executing a closed decision table, a new statement will be added to the language. Closed tables, which were described in sections 3.8.3 and 3.8.5, are analogous to internal subroutines. Execution of a closed table requires that control be passed to the start of the table, and that control be returned, after the last action has been executed, to the point at which execution was requested. Unlike FORTRAN subroutines, no parameters are passed to a closed table. The "PERFORM" statement is proposed for requesting the execution of a closed table. It would consist of PERFORM, followed by a statement number, either entered in columns 7 to 72 of a source record, or appearing as the action in an ACTION/MAPPING statement. Since a PERFORM can appear in a table, it is possible to nest calls to closed tables.

Several characters have been used for special purposes in writing decision tables; for example, the answer and mapping entry delimiter, don't-care, local else answer, and the symbols which appear as the answers of limited answer questions, and as the mapping entries of limited actions. It is possible to give the programmer a facility through which he can specify the character, or group of

characters, which he wishes to use for these special purposes. This might be done as an option when execution of the preprocessor is begun, or through a declaration in the program.

FORTRAN statements are usually coded in columns 7 to 72 of each source record. If the operating system under which the decision-table language is to be implemented encourages the use of records longer than eighty columns, then the language may be modified to take advantage. The primary benefit of longer source records is that fewer statements may require continuation. Answers and mapping entries are far easier to read when they are each coded on one statement. Unfortunately the width of the print line on the line printer which is used to produce the listing of programs, will still impose an upper limit on the width of the answers and mapping entries.

Since a preprocessor is to be built to translate decision-table programs into FORTRAN, it is possible to make changes and extensions to the FORTRAN language with little additional effort. A number of the modifications initially approved by the X3J3 Committee of the American National Standards Institute (ANSI) [21] could be made. Although these changes might improve the decision-table language and could prove convenient for the programmer, they are a contradiction of the design goal to produce a language as similar as possible to FORTRAN.

5.6 Limited Tables

It was suggested in section 5.2 that a simplified syntax could be developed for use with tables consisting entirely of limited answer questions and limited actions. The principal advantages of this syntax are that there is no need for a delimiter to separate either the answer or mapping rules, and the overall width of the rules is minimized. Only the abbreviated forms Y and N will be used for the limited answers. By using single characters it is possible to code all the answer rules on one source statement, and in many cases, to have sufficient room for the entire question as well. When continuation is required, it is proposed that only the question be continued onto successive source records. That is, if the question plus the answers will not fit on one record, part of the question, followed by all the answers, are placed on the first record, and the remainder of the question appears on additional records. The continuation records are identified by the presence of a character in column six. The restriction that the answers cannot be continued is reasonable since more than sixty answers can be coded without continuation, and a table of sixty rules is probably far too big to be practical. In fact, most authors state that a table of even twenty rules is too large for most people to comprehend, and should be broken up into at least two smaller

tables [3, pg. 166; 4, pg. 80].

Provided that the answers do not contain imbedded blanks, and that the question is separated from the answers by one or more blanks, then no special character is needed to mark the start or the end of the answers. As well, there is no need for the answers on each question in a table to begin at the same column, although this would make the answers far easier to read.

The mapping rule entries are also limited to one column each, and consist of an "X" to indicate execution, and either a blank or a character such as "-" to indicate a rule in which the corresponding action is not to be executed. If the blank is used for this purpose, then the start of the mapping entries must be marked by a character such as "|". Alternatively, the starting column could be specified by the programmer, or all the answers and mapping entries in a table could be placed in the same columns.

Writing a limited table in this simplified form does not require any further changes to the syntax which was developed in the previous sections, for the general form of a table. Both types of table can be coded in the same program, and each type can be used for either open or closed tables.

CHAPTER 6

Processor Design

In Chapter 5, a decision-table programming language was proposed, and a number of problems involved in its design were discussed. This chapter is concerned with the implementation of a processor which can translate programs written in the decision-table language into machine code. Reference will be made to an existing processor, called DET, which was designed and implemented as part of this research effort. Although the problems discussed in this chapter will be related to the language described in Chapter 5, most of these problems exist with almost any other decision-table programming language. The discussion of algorithms for generating code for a decision table will be deferred until the following chapter.

6.1 Design Approach

The major decision to be made concerning the implementation of a decision-table language processor is the type of processor. For a high-level language such as the one described in Chapter 5, three types of processor might be considered: compiler, precompiler, and interpreter. A compiler typically translates source language directly into

machine language. Precompilers, or preprocessors as they are often called, translate from source language into another high-level language (for which a compiler is available). The third type is an interpreter, which executes source language directly; that is, it does not produce a translation of the program, but rather, analyzes source statements each time they are executed.

Interpreters are used to process languages such as APL [22] and SNOBOL [23] for which it is very difficult to generate machine code. This is usually due to the facilities in the languages for making changes during execution, particularly as regards the data structures being used. The use of an interpreter incurs a major penalty, namely, very slow execution due to the processing which is required each time a statement is executed. Since there is no overwhelming difficulty in generating machine code for the decision-table language, there is no reason to use an interpreter.

The choice between precompiler and compiler is difficult since each type of processor has disadvantages. A major disadvantage of using a preprocessor is the extra level of overhead that is introduced by running both the preprocessor and the compiler. The features of the particular language to be implemented are the most important factor in determining the magnitude of this overhead. A measure of this overhead is the number of passes over the source program which each processor must make. A prepro-

cessor for the language described in Chapter 5 requires only one pass. Since FORTRAN compilers often use four to six passes, the cost of using both a preprocessor and a compiler may be only slightly higher than the cost for the compiler alone. By cost is meant the amount of computer resources used.

The greatest disadvantage of developing a compiler is its size and complexity. This will be apparent both during the implementation of a compiler, and afterwards in maintaining it. The knowledge and experience required by the personnel who build a compiler, plus time and resources used in designing, coding, and testing it, would present a formidable obstacle to many organizations. Their choice might have to be to develop a preprocessor, in spite of the additional computer resources which it will consume through its lifetime.

Another disadvantage in using a preprocessor is that two program listings, rather than one, must be handled by the programmer. The preprocessor will produce one listing, showing the decision-table source program, and the compiler will generate the other, showing the intermediate program. The latter will be of interest, since not all errors in the source program are likely to be detected by the preprocessor. Instead, some errors will be detected only by the compiler, and thus the appropriate error messages will appear in the second listing.

Unfortunately, use of a preprocessor forces one to accept the limitations of the available compiler and its run-time routines. One of the most severe limitations is the lack of facilities to aid the programmer when his program abnormally terminates. With the exception of a small class of compilers, including WATFIV [24, 25] and ALGOLW [26, 27], very little explicit information as to the cause and location of a program failure is made available to the programmer. Usually it is necessary to study a number of listings, including the source program listing, pseudo-assembler listing of the object code, loader or linkage editor map, and core dump, in order to determine which statement in the source language program was being executed when the run terminated.

Thus an advantage of developing a compiler, and its library of execution time routines, is that facilities can be included to assist the programmer in the event that his program fails. The first information that is needed is which source statement caused the failure and why. Determining why may require a substantial amount of work by the compiler (for example, to generate range tests for all subscripts). Also valuable would be a facility to eliminate the need for core dumps by printing a formatted list showing the names of the variables used in the program with their values at the time of termination.

Whether a preprocessor is a practical alternative

depends on the availability of a compiler for a language similar to the decision-table language. The language described in Chapter 5 was based on FORTRAN, and still strongly resembles it. As a result, a preprocessor to generate FORTRAN code is practical. In the rest of this chapter, which is concerned with implementation decisions and problems, it will be assumed that a precompiler is to be used to produce a FORTRAN program as the intermediate representation of a decision-table language program.

Because of the great cost in time and resources required to build a compiler, a cautious approach seems well-advised; for example, implement a preprocessor first. Although this would require some additional time and effort, the experience gained in producing the preprocessor, and the experience gained by its users, could lead to changes and enhancements to the language before the compiler is implemented.

Another decision to be made concerning the design approach is the selection of a programming language in which to code the preprocessor. The choice is probably between an assembly language and a high-level language. Since a preprocessor is relatively small and not too complex, a high-level language implementation is likely to be feasible. An assembler implementation should result in a faster-running preprocessor, but at the cost of a longer and more expensive development period. A sensible approach is to use

a high-level language first, and then, with the benefit of the experience gained in that implementation, to rewrite the preprocessor, or parts of it, in assembler.

6.2 Program Listing

The layout of the source program listing to be produced by the preprocessor presents some interesting problems. The central question is whether to produce one listing or two. Many programs will contain extended tables in which the answers on some questions have been continued. If these answers are to be readable in the listing, then the preprocessor must reformat them so that all the answers for any one question are printed on one line. On the other hand, a listing which shows the statements exactly as coded, is needed when changes must be made to a table.

Assuming that reformatting of statements outside tables is not useful, two possibilities may be considered: first, to print two complete listings of the program; and second, to print one listing which includes both the original and reformatted versions of each table. In the latter case, it is only necessary to print one version of any table that does not require reformatting.

An additional function of the reformatting should be to align the answer and mapping rules on each question and action in a table. This is also necessary for a table

to be easily readable. The alignment should be accomplished by inserting or deleting leading and trailing blanks in each rule entry. When necessary, imbedded blanks could also be deleted in order to squeeze all the rules onto a single line.

It is likely that many programmers would try to code each table with the rules neatly aligned and, when possible, without continuation. However, after part of a table has been coded, an unexpectedly long answer or mapping entry may ruin the alignment. Changes to a table after it has been coded are another opportunity for destroying alignment. It would be highly undesirable to have to repunch an entire table each time a change throws the alignment out, but this would be necessary if the preprocessor could not provide a reformatted listing.

Limited tables may also require reformatting. The starting columns of the answers and mapping entries in each table must be aligned. The width of a limited answer or mapping entry is fixed, and continuation is not permitted, so the reformatting may simply consist of shifting the rule entries on individual statements to a common starting column.

A much more serious problem with limited tables is that, due to the lack of delimiters, it may be difficult to determine in which rule a mapping entry, "X", appearing at the bottom of a table, is located. One solution is to use

some symbol, other than blank, (for example, a minus sign, "-",) to indicate "do not execute this action on this rule". Doing this produces a continuous string of characters for each rule. A better solution is to print the table in a box layout as shown in Fig. 6-1. This produces an uncluttered

J .EQ. 9	Y	Y	Y	N	N	N
K .LE. 256	Y	Y	N	Y	N	N
DIFF.LT..00001	Y	N	-	-	Y	N
ACTIONS						
J=J+1				X	X	X
K=K+64	X	X		X		
P=P*X+Q*F(P)		X		X		X
WRITE(6,77)J,K,P			X			
STOP 77			X			
PERFORM 25					X	
CALL SUBGOT	X					
GOTO 88	X			X		X
EXIT		X			X	

Fig. 6-1 Listing layout for a limited table.

and much more readable table.

In addition to the source progran listing, the preprocessor could also produce a cross-reference dictionary. In order to be able to refer to statements in the source program, the preprocessor could number the statements in the source listing. This number will be referred to as a "line number" in order to distinguish it from statement numbers coded by the programmer. The dictionary should consist of two parts: the first containing a sorted list of all statement numbers used in the program, and the second

part containing a list of all identifiers, in alphabetical order.

For each statement number, the line number at which it was defined, and the line number of each reference to it, should appear. The information for each identifier should include: first, its mode and dimension; second, the line number of any declaration in which it appears; and third, the line number of each other statement in which it appears, with an indication of which statements might change its value.

The cross-reference dictionary must be produced by the preprocessor and not by the compiler, because the entries in the dictionary should refer to the statements in the source program, not the intermediate program.

6.3 Debugging Aids

One of the primary objectives in writing a program is that it be "correct", that is, it does what it was designed to do, and is free of bugs. One method of approaching correctness is thorough testing. Since initial testing usually establishes that a program does contain bugs, it is reasonable to consider how to pinpoint their location. The following is a discussion of a number of debugging aids which the preprocessor can provide.

One type is assistance in locating where a program

failure occurred. Whether or not a table was being executed, which table, whether the questions were being evaluated or actions executed, and which rule was selected, could all be indicated. Since the preprocessor generates FORTRAN statements to represent decision tables, the debugging aids themselves must be implemented as FORTRAN statements. The code produced for the above indicators can consist of assignment statements placed at the beginning and end of the code for each table, and at the start of the code for each rule. To indicate which table is being executed, for example, assign the line number of the table statement to a variable dedicated to this purpose. If the program fails, the value of this variable identifies the table. The values of other variables would provide the other information. These values could be printed by the formatted-dump routine mentioned in section 6.1.

Another aid would be a list of, say, the fifty most-recently executed tables. The code required for this would use a vector of fifty elements, initially set to zero, and maintain a pointer to the next available element, initially the first. For each table, statements would be generated to store the line number of the table statement in the next element of the vector, and increment the pointer. Whenever the pointer value reached 51 it would be reset to one. Again, the formatted-dump routine could print the list in chronological order.

A count of the number of times each table was executed during a run could be useful in testing a program, for instance to indicate tables that had not been executed. To do this would require a vector containing an element for each table in the program. At the start of a run, each element would be initialized to zero. The code for each table would include a statement to increment the value for that table. In order to print these values at the completion of a run, the preprocessor might replace each STOP statement by a transfer to code to print the list. Provision would have to be made to avoid conflicts between such vectors generated in separate preprocessor runs. This same analysis could be extended to counting the number of times each rule in a table is executed during a run. The user might specify, through the use of a control statement, the tables to be monitored.

Debugging aids provided by the FORTRAN compiler could also be used by the decision-table language programmer. If such aids as a trace facility and subscript range checking are not available with the compiler, then their implementation in the preprocessor might be considered.

Each of the debugging aids suggested in this section would increase the cost of running a program. Both the CPU time required to execute the program, and the amount of memory needed for the object code, would be increased. The justification for using any of these aids is that the

cost of the computer resources required to provide the aids, is at least balanced by the value of their contribution to the correctness of the program and the time saved by the programmer.

6.4 Statement Numbers and Variables

In order to translate decision tables into FORTRAN statements, the preprocessor must introduce some statement numbers. Since the programmer also codes statement numbers, a convention must be established in order to avoid duplication. One possible convention is to restrict the programmer to numbers below 10000, and thus allow the preprocessor to generate any five digit number. A refinement of this idea is to allow the programmer to specify, in a control statement, the range of numbers which he will use, or alternatively, which he will not use. If the programmer mistakenly uses a number in the range allotted to the preprocessor, an error message could be produced by the preprocessor. Even if the preprocessor does not check for them, duplicate statement numbers should be detected by the FORTRAN compiler. Restrictions on the programmer's use of statement numbers, could be avoided with a two-pass preprocessor, by recording, on the first pass, the numbers used by the programmer. Then, during the second pass, the preprocessor could refer to this list in order to avoid generating

a duplicate.

A similar problem arises with identifiers. As was noted in section 6.3, several of the debugging aids require the use of dedicated variables to serve as counters and status indicators. In a later section, it will be shown that the PERFORM and EXIT statements also require dedicated variables. Obviously the programmer must not misuse such variable names. The possible solutions are analogous to those for avoiding duplicate statement numbers. The simplest convention is to begin or end each identifier generated by the preprocessor, with a particular group of characters, such as "ZZZ" or "TAB". Alternatively, the programmer could specify the characters. In any case, the programmer would avoid using any identifier with that prefix or suffix. It would be desirable for the preprocessor to detect any misuse by checking each identifier used by the programmer.

6.5 Nesting DO-loops and Tables

The appearance of DO-loops in decision tables leads to some interesting questions, particularly regarding the nesting of loops and tables. Tables themselves need not be nested since the PERFORM statement may be coded as an action when it is desired to execute a table, and return to the next action on the rule. The FORTRAN language allows

only one particular form of overlapping of DO-loops, namely where one loop is wholly contained within another.

There are three combinations of loops and tables which can be considered: first, one or more DO-loops may appear as actions in a table; second, a table may be contained within the range of a DO-loop; and third, a loop and a table may otherwise overlap. The third case would occur whenever one, but not both, of the DO statement and the final statement of the loop, appears as an action in a particular table. This overlapping case violates the definition of a decision table as being a complete unit, and is therefore not permitted.

In order to check for incorrectly nested DO-loops, and overlapping loops and tables, the preprocessor could maintain a stack. Whenever a DO statement is encountered, the statement number which marks the end of the loop is pushed onto the stack. When a statement number appears in columns one to five of a statement, the number is compared to all the numbers currently in the stack. If it matches the top number, then the loop has been ended correctly, and the number is popped from the stack. If it matches a number other than the top one, then incorrect nesting of DO-loops has been detected. If there is no match, then the statement number is not the end point of a loop. When either a TABLE or an EXIT statement is encountered, the stack will be empty, unless there are overlapping tables and loops. If

the FORTRAN compiler being used has a limit on the depth to which DO-loops can be nested, then the stack should be able to contain this number of entries. Otherwise the length of the stack must be chosen arbitrarily. If a DO statement is found when the stack is already full, an appropriate error message can be generated.

DO-loops present at least one additional problem. It concerns the FORTRAN statements which must be generated by the preprocessor when a DO-loop appears as an action in a table. If the loop is selected on more than one rule, then more than one copy of the loop may appear in the generated code. Thus, the preprocessor must replace the statement number which indicates the end of the loop, so that each occurrence of the loop uses a different number.

6.6 Closed Tables

Closed tables were included in the decision-table programming language in order to facilitate a modular programming style. The PERFORM statement was invented to execute a closed table as an internal subroutine. The following three sections contain a discussion of problems related to closed tables. In section 6.7, the FORTRAN code which must be generated for a closed table will be considered.

6.6.1 Recognition

The use of two different types of table in the same program raises the problem of how to determine the type of any given table. The preprocessor must be able to make this distinction, since different code will have to be produced for the beginning and end of each type of table. It would be best to make the type of each table apparent to anyone who reads the program. The most obvious way would be to include this information in the TABLE statement by coding either "CLOSED TABLE" or "OPEN TABLE".

Another problem is where a closed table may be placed in a program. The appearance of a closed table is a declaration since its execution can only be initiated by a PERFORM statement. In order to give the programmer freedom to declare a closed table near the area of the program where it will be used, it is proposed that the table may be placed anywhere except between the records of a continued statement. This definition raises a semantic question which is illustrated by the following segment of code:

```
   I=0
25 CLOSED TABLE
```

If the statement "I=0" is ever executed, then which statement should be executed next? One solution is to jump to the first executable statement following the closed table declaration. This is unsatisfactory since it would be easy

to mistakenly assume that the table would be executed next. An alternative is to require that the last executable statement preceding a closed table be an unconditional transfer.

6.6.2 Errors

The preprocessor must be able to detect several types of errors involving closed tables; first, PERFORM statements which reference statement numbers not appearing on closed tables; second, transfers to closed tables by statements other than PERFORM; and third, attempts to "fall into" a closed table.

The first of these errors can be detected in a one-pass preprocessor by building a list of statement numbers of closed tables. Each statement number referred to in a PERFORM statement, or defined in a closed table, is entered in the list. For each statement number there are two flags, one set when the table is found, and the other when a PERFORM referencing the table is encountered in the source program. After the entire program has been examined, any entry which has only one of the two flags set, is in error. Either the statement number appeared in a PERFORM, but was not defined on a closed table, or the number indicates a closed table which was never PERFORMed.

Illegal transfers to closed tables can be detected

in a one pass preprocessor, by building a list similar to the one described in the previous paragraph. The statement numbers entered in this list are those defined anywhere except on a closed table, and those referenced in any statement other than a PERFORM. When the end of the program is reached, undefined or unreferenced statement numbers will have only one flag set. If any of the undefined numbers appears in the closed-table list, then there have been references to closed tables by statements other than PERFORM.

Both of these error detection schemes have the disadvantage that error messages can only be produced at the bottom of the source listing, and indicate that an invalid use of a particular statement number occurred somewhere in the program. A major improvement would be to flag each offending statement where it appears in the source listing. Unfortunately, a two-pass preprocessor would be required to accomplish this. During the first pass, a complete list of statement number definitions would be built. Each reference to a statement number would then be checked during the second pass.

If the last executable statement preceding a closed table is not an unconditional transfer, then it is possible to fall into the closed table. In order to detect such errors, the preprocessor should check that only a GOTO, STOP, or RETURN statement appears in this position. Since

control cannot fall through a closed table, the first statement following a closed table can never be executed if it does not have a statement number. Any such statement should also be flagged by the preprocessor.

6.6.3 How to Exit

The definition of a closed table as an internal subroutine, implies that the only valid way to leave a closed table, is to return to the PERFORM which initiated execution of the table. Transfers out of a closed table, in the form of GOTOs, must be forbidden since there is no way to reenter the table to exit properly. Statements such as PERFORM and CALL are acceptable, however, since they include a return mechanism. A STOP statement might also be permitted, even though it is effectively a transfer statement (to return control to the operating system), since execution of the program cannot be restarted.

A problem arises with FORTRAN compilers which permit statement numbers to appear among the arguments in a subroutine call. The purpose of these statement numbers is to provide additional return locations. Their use is equivalent to executing a computed GOTO under the control of a value determined by the subroutine. Thus the use of such auxiliary returns must not be permitted on subroutine calls appearing in closed tables. It is questionable that their

use should be encouraged elsewhere.

Since a closed table is an internal subroutine, it is not possible to use the PERFORM statement to execute a table contained in a separate program or subroutine. Instead, the CALL statement is used for external routines. Suitable placement of ENTRY and RETURN statements will permit execution of a single table or a group of tables. Since external routines written in the decision-table language may include PERFORM statements, each routine must contain its own code to carry out the linkage between PERFORMs and closed tables.

One additional question is whether to permit a RETURN statement to appear as an action in a closed table. The effect of the RETURN would be to leave the subprogram immediately and return to the calling routine. Thus the normal return to the PERFORM statement would never be executed.

6.7 Code for PERFORM and EXIT

In order to allow the use of recursive calls to closed tables, the code generated by the preprocessor for PERFORM and EXIT statements could be based on a stack. Because recursion does not otherwise appear in the language, no provision has been made for a run-time stack. Although it is not difficult to provide a stack for closed table

linkage, there is an annoying problem, namely deciding on the length of the stack. Whatever length is chosen may be too small for some users and too large for others. One simple way to avoid the problem would be to have the user specify the length as an option when the preprocessor is run.

When a PERFORM is executed, the return address is pushed onto the stack, and when the EXIT statement of a closed table is reached, the return address can be popped from the stack. Since a series of successive PERFORMs, without corresponding EXITs, places a number of entries on the stack, the usual housekeeping work of checking for stack overflow, is required. The use of RETURN as an action in a decision table will result in code being generated to reinitialize the exit mechanism by emptying the stack.

Although generating FORTRAN statements to perform these stack manipulations is not difficult, the use of addresses as the stack elements is a problem since the FORTRAN language does not include a label mode. Two solutions to this problem will be presented, each of which is suitable for use with a one-pass preprocessor. The first solution requires that a list of references be built for each closed table. When the end of the program is reached, the lists are used to generate part of the code for the EXIT statements. The second solution requires two generalizations of the definition of the ASSIGN and the assigned GOTO

statements.

6.7.1 Computed GOTO Code

The first solution is to avoid placing addresses on the stack. Instead, integer values are stacked or unstacked when a PERFORM or EXIT statement is executed. The integer values are used as indexes in computed GOTO statements, one of which is generated for each closed table. The statement numbers appearing in the computed GOTO for a particular closed table, are the return addresses for all the PERFORM statements which reference that table.

When a PERFORM statement is encountered, a value is pushed onto the stack. The value is "1" if this is the first PERFORM to reference this particular closed table, "2" for the second reference, etc. After stacking the value, a transfer to the table is generated. The next statement is the location to which control will be returned; if it does not have a statement number, then one is generated. In either case, the statement number is stored by the preprocessor in the list of references to that table.

The code for an EXIT statement must pop the top value from the stack, and use it as the index of a computed GOTO which lists all the possible return points for that table. Unfortunately, in a one-pass preprocessor, not all the return points would necessarily be known when the table

itself appears in the program. One way to avoid this problem is to require that a closed table appear only after all PERFORM statements which reference it. A better solution is to delay generation of the computed GOTO statements until the end of the program has been reached, when all the return points will have been noted. The code actually generated for the EXIT can include a transfer to the computed GOTO. The following example shows, on the left, decision-table language statements and, on the right, corresponding FORTRAN statements as they might be generated by the preprocessor:

```
    .
    .
    .

PERFORM 25                      ZZZPT=ZZZPT+1
                                IF(ZZZPT.GT.ZZZTOP)GOTO 99991
  .                             ZZZSTK(ZZZPT)=1
  .                             GOTO 10001
  .                       10002 CONTINUE
                                  .  .  .
PERFORM 25                      ZZZPT=ZZZPT+1
                                IF(ZZZPT.GT.ZZZTOP)GOTO 99991
  .                             ZZZSTK(ZZZPT)=2
  .                             GOTO 10001
  .                       10003 CONTINUE
                                  .  .  .
PERFORM 40                      ZZZPT=ZZZPT+1
                                IF(ZZZPT.GT.ZZZTOP)GOTO 99991
  .                             ZZZSTK(ZZZPT)=1
  .                             GOTO 10004
  .                       10005 CONTINUE
                                  .  .  .
PERFORM 25                      ZZZPT=ZZZPT+1
                                IF(ZZZPT.GT.ZZZTOP)GOTO 99991
  .                             ZZZSTK(ZZZPT)=3
  .                             GOTO 10001
  .                       10006 CONTINUE
                                  .  .  .
```

```
40 CLOSED TABLE                    GOTO 99992
     .                          40 GOTO 99993
     .                       10004 CONTINUE
     .                             .  .  .
   EXIT                            ZZZRET=ZZZSTK(ZZZPT)
     .                             ZZZPT=ZZZPT-1
     .                             GOTO 10007
     .                             .  .  .
25 CLOSED TABLE                    GOTO 99992
     .                          25 GOTO 99993
     .                       10001 CONTINUE
     .                             .  .  .
   PERFORM 40                      ZZZPT=ZZZPT+1
                                   IF(ZZZPT.GT.ZZZTOP)GOTO 99991
     .                             ZZZSTK(ZZZPT)=2
     .                             GOTO 10004
     .                       10008 CONTINUE
                                   .  .  .
   EXIT                            ZZZRET=ZZZSTK(ZZZPT)
     .                             ZZZPT=ZZZPT-1
     .                             GOTO 10009
     .                             .  .  .
   END                     10009 GOTO(10002,10003,10006),ZZZRET
                           10007 GOTO(10005,10008),ZZZRET
                                 END
```

In this example, all the statement numbers generated by the preprocessor are greater than 10000, and the variable names introduced by the preprocessor are prefixed by "ZZZ". Declarations for the variables, and the code for statement numbers 99991, 99992, and 99993, are omitted. This code would handle the three errors which could be detected during execution: first, stack overflow; second, an attempt to "fall into" a closed table; and third, an attempt to transfer to a closed table by use of a statement other than PERFORM. The four variables introduced by the preprocessor are: ZZZSTK, a vector which acts as the stack; ZZZPT, a pointer to the top entry on the stack; ZZZTOP, a pointer

to the highest location in the stack, which is used to test for stack overflow; and ZZZRET, which is used to satisfy the FORTRAN language requirement that the index of a computed GOTO must be an integer variable.

It has already been noted that this method of generating code for closed tables is less than ideal since a list of references to each closed table in a program must be built. Another serious objection is the detrimental effect of this code in a virtual memory environment, where movement from page to page must be minimized. Since the computed GOTO statements for the closed tables are placed at the end of the program, it is quite likely that some closed tables will appear in different virtual pages than their computed GOTOs. Thus the return from a closed table will often involve a paging operation which would not be needed if the code for an EXIT were generated contiguously.

6.7.2 Assigned GOTO Code

The second solution to the code generation problem avoids the two faults of the previous solution, but requires two modifications to the FORTRAN language specifications concerning the ASSIGN and assigned GOTO statements. The first modification involves the sentence, "Once having been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned

GOTO statement until it has been redefined . . . " [28, pg. 599]. The change is to allow an integer variable, which has been mentioned in an ASSIGN statement, to appear in an assignment statement. The intent of this change is to permit the value stored in an integer variable by an ASSIGN statement to be assigned to a different variable or an element of an array. The second modification is the addition of an alternate form of the assigned GOTO statement namely "GOTO i", where "i" is an integer variable. The change involved is the deletion of the list of statement numbers which this GOTO can transfer to.

Neither of these changes should cause major difficulty for the FORTRAN compiler implementor. The value stored by an ASSIGN statement is typically an address, which can be manipulated as though it were an integer value. The list of statement numbers which has been deleted from the assigned GOTO statement is not useful for generating code.

If these two changes were made, the following code could be produced by the preprocessor for a PERFORM statement:

```
      ZZZPT=ZZZPT+1
      IF(ZZZPT.GT.ZZZTOP)GOTO 99991
      ASSIGN 10002 TO ZZZRET
      ZZZSTK(ZZZPT)=ZZZRET
      GOTO 10001
10002 CONTINUE
```

This code is the same as that used with the computed GOTO in

the previous section, except that an actual return address is placed on the stack. An additional change to the ASSIGN statement to permit use of a subscripted variable would shorten the code by replacing the third and fourth statements by:

```
ASSIGN 10002 TO ZZZSTK(ZZZPT)
```

No change is required in the code generated at the start of a table. The statements produced for the EXIT from a closed table would be the following:

```
ZZZRET=ZZZSTK(ZZZPT)
ZZZPT=ZZZPT-1
GOTO ZZZRET
```

This code is very similar to that used in the previous section. The major difference is that no additional statement is needed at the end of the program to complete the code for each EXIT. Another change to the assigned GOTO statement to allow the use of a subscripted variable, would shorten this code to the following:

```
ZZZPT=ZZZPT-1
GOTO ZZZSTK(ZZZPT+1)
```

The code shown in the examples in this section and the previous one, could be shortened by coding the statements which increment the stack pointer, and check for stack overflow, at the start of each closed table rather than for

each PERFORM statement. The following example illustrates this variation using the ASSIGN and assigned GOTO statements:

```
      •
      •
      •
   PERFORM 30                    ASSIGN 10002 TO ZZZRET
     •                           GOTO 10001
     •                       10002 CONTINUE
     •                             •  •  •
   PERFORM 30                    ASSIGN 10003 TO ZZZRET
     •                           GOTO 10001
     •                       10003 CONTINUE
     •                             •  •  •
30 CLOSED TABLE                  GOTO 99992
                              30 GOTO 99993
     •                       10001 ZZZPT=ZZZPT+1
     •                           IF(ZZZPT.GT.ZZZTOP)GOTO 99991
     •                           ZZZSTK(ZZZPT)=ZZZRET
                                   •  •  •
   EXIT                          ZZZRET=ZZZSTK(ZZZPT)
     •                           ZZZPT=ZZZPT-1
     •                           GOTO ZZZRET
     •                             •  •  •
   END                           END
```

6.8 Detecting Errors

One of the major reasons for using decision tables is that the computer can check them for errors. Unfortunately for the person who implements a preprocessor, the error checking must be done by the preprocessor.

The foremost question is whether or not to design the preprocessor to detect even those errors which would be detected during the subsequent FORTRAN compilation. A

reason for detecting these errors during preprocessing is that it will minimize the need for the programmer to study a second source listing, namely the one produced by the compiler. In addition to the nuisance of working with two listings, the compiler listing is awkward to use in that the programmer must determine which source statement caused a particular intermediate statement to be generated. The opposite view is that detecting such errors in the source program requires a great deal of work in the preprocessor. Most of it is a complete duplication of the error checking which will be done by the compiler. The answer to this design question depends on whether the cost of implementing thorough error checking can be considered as an investment which will be repaid by the higher productivity of the programmers using the preprocessor.

Even if the decision is not to search for errors which will be found by the FORTRAN compiler, the preprocessor must still thoroughly check each decision table. The following three sections describe the checking that is required for each table. The preprocessor can help the programmer to see the correspondence between source and intermediate statements. For example, comments can be placed in the intermediate program to show where the code for each table begins and ends.

One additional point which may influence the decision on error checking, is whether a cross-reference

dictionary is being produced by the preprocessor. If so, each statement of the source program must be scanned to locate each usage of an identifier or statement number. In this case, it may require only a relatively small additional effort to detect most errors.

The form of the error messages produced by the preprocessor is another interesting question. One possibility is to print a code number immediately after any statement which is in error. The programmer would look up the code in a reference manual to determine the meaning of the message. This approach is ideal for the preprocessor, since it minimizes the space required for storage of messages, and simplifies the procedure for the generation of messages. Unfortunately, it is also the least helpful form of message for the programmer. From the programmer's point of view, it would be ideal to get detailed messages, free of abbreviations and codes, and indicating the specific characters which are at fault. A problem is that such messages require substantial storage space. In a preprocessor written in a high-level language such as FORTRAN, it is unlikely that a minimum-space text optimization technique such as the one described by Wagner [29], would be practical. An alternative is to place the error messages in a direct-access file on an auxiliary storage device. The advantage gained by avoiding cryptic messages should outweigh the costs involved in the storage and access of the messages.

6.8.1 Preliminary Checks

A one-pass preprocessor reads the entire source program once, but each table requires additional passes for error checking and code generation. Thus intermediate storage of the statements which form a table is required. The choice of storage medium is largely determined by the environment in which the preprocessor will be run. Main memory is probably the most cost-effective form of storage, if it is available, since the overhead of using the operating system routines to access auxiliary storage is avoided. At least the rules of the table should be kept in main memory, if at all possible, because of the number of passes over them which will be required for the sort described in the following section. The remainder of this section will be centered on the preliminary error checking which can be performed during the first pass over a table.

The preprocessor detects the start of a table by the presence of a TABLE statement. Whether the table is open or closed is also indicated by the TABLE statement. The latter information is used both to ensure that illegal transfers out of a closed table are not made, and to generate the correct code at the start and end of the table. Analysis of the first QUESTION/ANSWER statement indicates whether limited or extended format is used, and also the number of rules. The appearance of the delimiter character,

"|", indicates that an extended format table has been found. Although an extended format table can include both limited and extended answer questions, they are distinguishable by the absence or presence, respectively, of the underscore character, "_", which marks the location of the missing portion of an extended answer question.

If thorough error checking is being done, or a cross-reference dictionary is being built, then each question must be analyzed to detect syntax errors, or to add each reference to an identifier or statement number to the dictionary. For this analysis, an extended answer question is treated as though it were several questions, each one formed by replacing the underscore with a different answer. The answers of limited questions are checked for validity, and the number of rules should not vary from statement to statement in a table. Any question whose answers are entirely don't-cares, should be flagged with an error message, since it does not participate in the selection of a rule. No QUESTION/ANSWER statement may have a statement number.

The ACTIONS statement marks the transition from questions to actions. Each ACTION/MAPPING statement is checked for valid syntax, correct number of rules, and proper mapping entries. If FORMAT statements are encountered among the ACTION/MAPPING statements, they can be written into the generated FORTRAN code immediately after

they are checked for errors. Section 6.5 includes a description of a technique for detecting overlapping DO-loops. Any statement number on an ACTION/MAPPING statement which does not mark the end of a loop is invalid.

Several additional checks can be made on the mapping entries of each table. Each ACTION/MAPPING statement should have at least one non-blank mapping entry; otherwise, the action can never be executed. If such a statement appears in a table, the programmer has probably forgotten to code mapping entries. Although limited actions, which are executed on each rule, are quite valid, they should appear only when the action must be executed in a particular sequence with the other actions in the table. In order to indicate where control will be passed after execution of an open table, the last (lowest) action selected on each rule must be either EXIT, GOTO, STOP, or RETURN. If one of the latter three actions is selected on a rule, then no action below it should be selected since the lower action would never be executed. The preprocessor should also warn the programmer if any two or more rules in a table have identical sets of mapping entries. In such a case the programmer may be able to combine these rules by the introduction of don't-cares.

During the first pass over a table, the maximum widths used for the answers on each rule can be computed. This information is needed to determine whether all the

rules can be printed on one line. If not, the table is rejected, and a message is produced to indicate that the table should be divided into at least two smaller ones.

6.8.2 Ambiguity and Completeness

When the end of a table, marked by an EXIT statement, is reached, the next phase of error checking can be performed. This is to check the table for ambiguous or incomplete combinations of answers. A table is ambiguous when for some combination of answers, more than one rule can be selected. The opposite case is an incomplete table, where for some combination of answers, no rule can be selected.

The technique proposed to accomplish these checks, is a sorting of the rules, using the answers as the key. As the sorting progresses through the table, any question whose answers are the cause of ambiguity or incompleteness in the table is detected.

Consider first the sort for a limited table, where the only valid entries are yes, no, and don't-care. The first step is to order the rules such that all the rules containing yes as the answer to the first question, precede the rules with no. The first question cannot have don't-cares; if any are present then an ambiguous table has been found. Ordering of rules means that when two anwers must be

interchanged, the entire rules containing these answers are interchanged. After the ordering, the answers to the first question appear as two contiguous groups, the first containing all the yes answers, and the second the no answers. If there is only one question in a table, then the sorting is complete after the first step.

When there are two or more questions, step two is performed for each remaining question. Step two is to order the answers within each group of rules, so that all the rules containing yes as the answer, precede those with no, which in turn precede those with don't-cares. The groups are those established by the ordering of the answers on the preceding question. To be valid, a group must contain either of two combinations: first, both yes and no answers, but no don't-cares; or second, only don't-cares. All other combinations are erroneous. If only yes or no but not both, appears, then the table is incomplete. When yes and no and don't-cares all appear, then the table is ambiguous. A table can be both ambiguous and incomplete, if a group contains don't-cares as well as either yes or no. Each valid group which contains yes and no answers, is divided into two groups for the sort of the following question.

The answers on the final question of a table should consist of either a single don't-care, or one yes and one no, in each group; otherwise, the table is ambiguous.

A short example will illustrate this technique.

The following is the answers section of a limited table containing three questions:

```
YYNNYY
NN--YY
NYYNNY
```

The first step is to order the answers of the first question to produce:

```
YYYYNN
NNYY--
NYNYYN
```

The answers of the first question now form two groups. Step two is to order the answers of the second question in the two groups of rules. In this example, only the first group actually requires ordering, while the second group remains unchanged:

```
YYYYNN
YYNN--
NYNYYN
```

As a result of this last ordering, there are now three groups of rules. The final step is to order these three groups of answers of the third and last question:

```
YYYYNN
YYNN--
YNYNYN
```

The sort for an extended table will now be considered. It is slightly more complicated because of

several of the features of extended tables, such as the generality of the extended answers, the potential use of global and local else answers, and the two types of question. The global else rule does not take part in the sort, but retains its position as the rightmost rule in the table. Although extended answers may contain imbedded blanks, these blanks are not significant in determining whether answers are identical. Therefore, all imbedded blanks should be deleted from extended answers before doing comparisons on them.

The answers of a limited question can be handled exactly as in a limited table. When ordering the answers of an extended question, the answers are placed in ascending order. The local else answer, if it is present, participates in the sort as though it were merely another extended answer. In order for a group of extended answers to be valid, it must consist of one of two combinations of answers: first, at least one extended answer, with or without a local else answer, and no don't-cares; or second, only don't-cares. An extended question is never considered to be incomplete since the else answers handle all cases where the actual answer to a question is not one of those specified. Whenever a local else answer is not contained in a group of extended answers, the preprocessor will generate code such that the global else rule takes its place. If a table does not include a global else rule, then the

preprocessor will add one. The only action to be executed on this rule would be an error halt, with an appropriate error message.

The following pair of questions illustrates an interesting use of extended answers:

K.EQ._	7	7	7	7	7	8	8	8	9	9	9
M.EQ._	1	2	3	4	?	1	2	?	2	4	?

Here the answers for the second question are not identical in each group. The reason is that the local else answer has been used with the meaning "everything else" or "all other values". This allows great flexibility since the programmer need only code those answers for which there is a unique set of actions to be executed. Unfortunately, if the programmer forgets to code an answer, there is no way for the preprocessor to detect the omission.

Some time saving may be obtained in the sorting if, instead of actually interchanging rules, a vector of pointers to the rules is maintained, and only the pointers are interchanged. Every reference to the rules would then be done indirectly via the pointers.

6.8.3 Semantic Errors

There is another class of errors which do not involve violation of the syntax rules of the language, but which are a serious problem. Consider the following extended question:

```
I .GT. _      | 1 | 2 | 3 |
```

For values of I greater than two, this question is ambiguous since more than one rule should be selected. If code were to be generated for this table, only the order of testing for each of the three answers would determine the rule selected.

Another potentially ambiguous extended question is the following:

```
M .EQ. _      | J | K | L |
```

Whether or not this question is ambiguous, is determined by the values of M, J, K, and L, when the question is evaluated. If any two of J, K, and L are equal, and are also equal to M, then the question is ambiguous.

In both these cases the programmer could avoid these ambiguities by recoding the questions. The second example might be written as:

```
M .EQ. J      YYYNNNN
M .EQ. K      YNNYYNN
M .EQ. L      -YNYNYN
```

This reformulation of the table could have been done by the preprocessor; however, if the number of answers in the original version of the question had been higher, for example twelve, then the number of rules in the revised table may be considered prohibitively large.

It is quite possible that the programmer is sure that a group of variables will always have different values, and that a question such as the second example above will never be ambiguous. For example, consider the following statements which might appear in a program:

```
INTEGER COLOUR,RED,WHITE,BLUE
DATA RED/1/,WHITE/2/,BLUE/3/
  •
  •
  •
READ(5,20) COLOUR
  •
  •
  •
TABLE
COLOUR .EQ. _     | RED | WHITE | BLUE |
  •
  •
  •
```

The programmer has wisely used meaningful names instead of numerical values for the three codes. Unfortunately, even this table could be ambiguous at execution time if one of the colour names is changed. If any of the names RED, WHITE or BLUE appears as an argument in a subroutine call, or in a

COMMON or EQUIVALENCE statement, then such a change could occur inadvertently.

It is apparent that ambiguities in extended tables cannot always be detected during preprocessing. In view of this situation, several courses of action might be considered:

i) Eliminate execution time ambiguities by severely limiting the form of extended questions. Possibly allow only a test for equality between a variable and a group of constants.

ii) Assume that extended tables will never be ambiguous at execution time, and therefore make no attempt to check for such an eventuality.

iii) Assume that ambiguities may appear, and generate code to detect them, regardless of the overhead involved.

iv) Compromise between the last two approaches, by making generation of the checking code optional. One possible strategy would then be to use the checking code during the testing of a program, but remove it for production runs of thoroughly tested programs.

The first of these suggestions would severely limit the usefulness of extended tables. Even the above example using names for colour codes would be ruled out. The compromise has the major disadvantage that it could give programmers a

"false sense of security". They might feel that all ambiguities would be detected. The remaining two alternatives appear to be better choices: either the programmer is responsible for preventing ambiguities, or the overhead of execution time checking is incurred.

6.9 DET

DET is an acronym for DEcision Table programming language. As part of this research effort, the language was designed and a preprocessor was implemented. The language described in Chapter 5 is more powerful and was designed after experience was gained with DET. The Appendix of this thesis consists of the programmer's reference manual for DET. Since this manual describes the implementation in some detail, only the major points will be mentioned here.

The one noteworthy restriction in DET is that only limited, open tables are permitted. Like the language in Chapter 5, DET is based on FORTRAN and bears a very strong resemblance to it.

The preprocessor, which requires one pass over the source program, was implemented in FORTRAN and generates a FORTRAN intermediate program. It consists of about 800 statements, a large portion of which are used in error checking. The technique described in section 6.8.2 was used to check each table for ambiguities and completeness. The

sort algorithm used was a transposition sort [30]. This type of sort was selected since no storage is required beyond that needed for the values being sorted. Since a programmer may often code answers in the required order, and since the answers tend to be repeated, a transposition sort is well suited because it takes advantage of inherent ordering and duplication in the values being sorted.

Although each decision table is thoroughly checked for errors, no attempt is made to locate errors which the FORTRAN compiler will detect when the intermediate code is compiled. About fifty different error messages can be produced.

CHAPTER 7

Code Generation

This chapter is a continuation of the discussion begun in the previous chapter, on the subject of implementation. Its purpose is to demonstrate that decision tables can be readily translated into executable code, or at least into a high-level language program which can then be compiled with existing software. In the majority of examples which appear in the following sections, the decision-table programming language described in Chapter 5 will be used, and FORTRAN statements will be generated.

The topic of code generation for decision tables has attracted the interest of a number of researchers. The "tree" and "rule mask" methods were proposed in 1962 and 1965, respectively, and have been the basis for almost all the work reported to date. In the following sections, these two techniques, and two others, will be introduced, and a brief discussion of their advantages and disadvantages will be given. The final section of this chapter deals with the problem of side effects.

7.1 Techniques

The primary criteria for judging the efficiency of a code-generation technique are: first, the quantity of storage required for the generated code; and second, the amount of time used to execute the code. The degree of complexity of the technique will influence both the effort required to implement it, and the resources used in running the processor. In most situations this latter point will be of secondary importance.

7.1.1 Repetitive Testing

Before considering the tree and rule mask methods of code generation, a "quick and dirty" technique will be described. Although the code produced is rather inefficient, in terms of both storage and execution time, the method is of interest because of its simplicity of implementation. It is also useful since it provides a reference point from which to compare the improvement which can be achieved with the other methods.

This technique, which will be referred to as the "repetitive testing" technique, requires separate testing for each rule in a table. The rules are processed one at a time, perhaps from left to right, and for each rule, all the relevant questions are evaluated. As soon as a rule is found whose answers match the results of the questions, the

actions selected on that rule can be executed. In order to illustrate this method with a specific example, the following limited table will be used:

```
TABLE
I .EQ. 9              YYNNN
P .GE. 127.43         --YNN
CNT .LT. 100          YN-YN
ACTIONS
I=1                   XX
WRITE(6,125)R           X
STOP                    X
I=I+1                    XX
GOTO 99               X  X
EXIT                   X  X
```

The FORTRAN statements generated by a preprocessor using the repetitive testing technique, might resemble the following:

```
C     BEGIN CODE FOR TABLE #1.
      IF(I.EQ.9.AND.CNT.LT.100)GOTO 10001
      IF(I.EQ.9.AND..NOT.(CNT.LT.100))GOTO 10002
      IF(.NOT.(I.EQ.9).AND.P.GE.127.43)GOTO 10003
      IF(.NOT.(I.EQ.9).AND..NOT.(P.GE.127.43).AND.
     *   CNT.LT.100)GOTO 10004
      IF(.NOT.(I.EQ.9).AND..NOT.(P.GE.127.43).AND.
     *   .NOT.(CNT.LT.100))GOTO 10005
      GOTO 99994
10001 I=1
      GOTO 99
10002 I=1
      GOTO 10006
10003 WRITE(6,125)R
      STOP
10004 I=I+1
      GOTO 99
10005 I=I+1
      GOTO 10006
10006 CONTINUE
C     END OF CODE FOR TABLE #1.
```

The statement "GOTO 99994" is included in the code to handle the case in which none of the rules is selected.

If the answers have been checked for errors, as described in section 6.8, then this statement should never be executed. Nevertheless, such a statement is valuable during the testing of a preprocessor since it could detect an internal bug which might otherwise remain hidden. The error could be either generation of incorrect IF statements or the failure to detect an error in the answers. Since large programs are seldom fully debugged, such error detecting statements should never be deleted.

Code such as a GOTO statement which transfers control to the statement immediately following the GOTO, can be eliminated, but at the cost of some additional checking by the preprocessor. Since the main reason for using this code generation technique would be its simplicity, the extra effort to detect such specific cases can hardly be justified.

The most serious inefficiency of the code produced with this technique is the amount of CPU time used when one of the last rules to be tested is selected. On the other hand, only one IF statement is executed when the first rule is selected. This leads to the conclusion that the average time required for testing can be minimized if the relative frequencies of rule selection are known. With this information, the tests can be ordered so as to test first the rules most likely to be selected. Of course if the frequencies were evenly distributed across all the rules,

there would be little to gain. This idea of using relative frequencies will be discussed more fully in the following section.

The quantity of storage used for the tests is another drawback. Fortunately this quantity can be reduced by saving the result obtained from evaluating the questions, and testing only the results. Thus each test consists of checking whether a logical variable is true or false. If some of the questions are long and complex, this strategy should substantially shorten the storage required. Using this modification, the following FORTRAN code might be generated:

```
      LOGICAL ZZZQ01,ZZZQ02,ZZZQ03
        .
        .
        .
      ZZZQ01=I.EQ.9
      ZZZQ02=P.GE.127.43
      ZZZQ03=CNT.LT.100
      IF(ZZZQ01.AND.ZZZQ03)GOTO 10001
      IF(ZZZQ01.AND..NOT.ZZZQ03)GOTO 10002
      IF(.NOT.ZZZQ01.AND.ZZZQ02)GOTO 10003
      IF(.NOT.ZZZQ01.AND..NOT.ZZZQ02.AND.ZZZQ03)GOTO 10004
      IF(.NOT.ZZZQ01.AND..NOT.ZZZQ02.AND.
     *   .NOT.ZZZQ03)GOTO 10005
        .
        .
        .
```

In programs to be used for lengthy production runs, an expensive translation to machine code is likely to be a good investment. The efficiency of the resulting code

should produce an overall saving in the cost of using the program. However, in a teaching environment, or during the testing and debugging of a program, a fast and inexpensive translation may be a better choice. Slow execution and large storage requirements are not significant when only a relatively small proportion of the time is spent executing the object code. The repetitive testing technique is practical only in the latter cases, where the inefficiency of the code produced can be tolerated, and the extreme simplicity of the technique is advantageous.

7.1.2 Tree

Use of the tree method for generating code results in more efficient code than the repetitive testing method, but at the cost of having to implement a more complex algorithm in the processor. This method produces code in the form of a tree structure of tests of questions. The name "sequential testing procedure" (STP) has often been applied to the resulting network of tests. The efficiency of this code ensues from the fact that a rule can always be selected after one test of each relevant question.

When this method is used on a limited table, a binary tree is produced. The code generation procedure involves the following steps: first, a test is generated for the top question in the table; second, two subtables are

formed, containing the remaining rules when the top question and its answers are deleted; and third, the procedure is repeated on the subtables. The first subtable contains those answers which fell under the affirmative of the preceding question, and the second contains those under the negative. The procedure terminates when a subtable is formed which consists of a single rule. The actions for that rule can then be generated. The following FORTRAN statements might be produced using the tree technique, for the table in the preceding section:

```
      IF(I.EQ.9)GOTO 10001
      IF(P.GE.127.43)GOTO 10002
      IF(CNT.LT.100)GOTO 10003
      GOTO 10004
10001 IF(CNT.LT.100)GOTO 10005
      GOTO 10006
        .
        .
        .
```

The statement numbers 10002 through 10006 begin the actions for rules 3, 4, 5, 1, and 2, respectively. The code for the actions would correspond to that shown in the previous section.

Even though there are still four IF statements in this code, there has been a substantial improvement since only four instead of twelve tests were generated. More important is that a maximum of three tests need be executed for the selection of any rule. In the code for repetitive

testing, only one of the five rules could be selected after three tests.

If a processor is written in a language which includes recursion, then the tree method of code generation can be implemented quite easily. Otherwise, the recursive procedure described above could be achieved by maintaining an explicit stack to hold pointers to the rules yet to be processed as each branch of the tree is developed. This is the technique used in the DET preprocessor which was briefly described in section 6.9. As it has been defined, this procedure requires the prior execution of a sort similar to that discussed in section 6.8.2. Without a sort, a rather complex algorithm would be required to form subtables from the unordered answers.

The tree method can also be applied to extended and mixed tables. The only apparent difference is that instead of a binary tree, code is produced in the form of an n-ary tree. The number of branches depends on the number of answers which are specified for each question. Below are two extended answer questions:

```
K .EQ. _    | 7 | 7 | 7 | 8 | 8 | 9 |   |
N .EQ. _    | 1 | 2 | 3 | 2 | ? | - |   |
```

The following FORTRAN statements might be generated for these extended questions. Notice that both a local else answer (in rule five) and a global else answer (rule seven)

appear in this example:

```
      IF(K.EQ.7)GOTO 10001
      IF(K.EQ.8)GOTO 10002
      IF(K.EQ.9)GOTO 10003
      GOTO 10004
10002 IF(N.EQ.2)GOTO 10005
      GOTO 10006
10001 IF(N.EQ.1)GOTO 10007
      IF(N.EQ.2)GOTO 10008
      IF(N.EQ.3)GOTO 10009
      GOTO 10004
        .
        .
        .
```

The code for the actions is not shown in this example. Statement numbers 10003 through 10009 correspond to rules 6, 7, 4, 5, 1, 2, and 3, respectively.

The tree technique of code generation first appeared in 1962, in an article by Montalbano [31]. He introduced two variants of the method. The first was designed to minimize the amount of storage needed for the questions. To do so, tests are generated in an order which isolates rules as soon as possible. The second variant was to minimize the average number of tests executed. Here, the opposite strategy is used, that is, to delay isolating rules as long as possible. This is done by generating tests in an order which keeps the subtables as close as possible to equal in size.

This first article seemed to kindle the interest of other researchers, and several more articles soon appear-

ed. Egler [32] proposed a very simple modification to the tree method. He suggested that efficient code could be produced by generating tests in an order determined by the number of don't-cares appearing in the answers of each question. Montalbano [33] demonstrated that this procedure was not reliable, and pointed out that, worse, an else answer would be awkward to handle.

In 1965, articles by Press [34] and Pollack [35, 36] were published, and contributed minor refinements of Montalbano's ideas. Reinwald and Soland [37, 38] presented two more general algorithms which they guarantee will produce: first, code with minimum average processing time; and second, code with a minimum storage requirement. Finally they showed a method for combining the two algorithms to obtain minimum overall cost. Unfortunately, these algorithms require specification of both the cost of executing each test, and the relative frequency of selection for each rule. Neither of these types of information is likely to be conveniently available.

Rule frequencies may occasionally be known, or may be guessed by the analyst or programmer. Often they can only be determined by monitoring the selection of rules as the program is run on live data. Although accurate rule frequencies are essential to obtain minimum cost, the costs of determining the frequencies have not been considered. The cost of executing a test may also be difficult to

ascertain, particularly in a preprocessor. The programmer is not likely to be in any better position to estimate these costs. In fact, it would be undesirable to concern the programmer with the structure of the code which will eventually be generated for the tables which he codes.

7.1.3 Rule Mask

The second major code generation technique is usually referred to as the rule mask method. It was first proposed in 1965 by Kirk [39]. The method requires that a matrix containing the answers for a table be part of the code generated for the table. When the table is executed, all the questions are evaluated, and a vector is formed containing the actual answers to the questions. This vector is then compared to each column (rule) of the answer matrix. When a match is found, the actions for that rule can be executed. In performing the comparisons, a don't-care matches any answer to the corresponding question.

To illustrate the operation of this method, consider the matrix of answers for the table given in section 7.1:

```
| Y Y N N N |
| - - Y N N |
| Y N - Y N |
```

If the value of the three variables appearing in the

questions were I=6, P=130.5, and CNT=88, then the following vector of answers would result from evaluating the questions:

N
Y
Y

When this vector is compared to the first column of the answer matrix, it is found that they do not match because the row one values differ. Similarly, the second column does not match. Comparison with the third column does produce a match since the don't-care in the third row matches either yes or no. Thus the actions selected on rule three would be executed.

There is one obvious advantage of the rule mask method over the tree method, namely that each question appears only once in the code generated for a table. This advantage is partially offset by the need to store the matrix of answers. For limited tables, only two bits per answer are required, since there are just three states to be represented: yes, no, and don't-care. Extended tables would require at least four bits per answer, in order to be able to represent a don't-care, local else answer, and several different extended answers. A subroutine could be linked to the object program to provide the code to perform the checking of answers. This would mean that instead of having to generate these instructions in the code for each table,

only a subroutine call would be needed.

There is also a serious disadvantage, namely the amount of CPU time needed, both to evaluate all the questions, and to match the answers. If the relative rule frequencies were known, then the time used in the matching process could be minimized by ordering the columns so that the most frequently selected ones are tested first. The comments about obtaining rule frequencies, which were made in the preceding section, apply here also.

An improvement to the rule mask method, which allows faster execution at the cost of some additional storage, has been suggested by King [40]. He assumes that rule frequencies are available, and proposes that only those questions relevant to the most freguently selected rules, be tested at first. Then, if these answers match, some time has been saved. Otherwise, additional questions are evaluated so that more rules can be matched. The extra storage contains a list showing the order of evaluation of questions and matching of answers.

An article by Muthukrishnan and Rajaraman [41, 42], points out that checking for ambiguity at execution time can very easily be accomplished with a slight modification to Kirk's method, but not with King's alteration. They propose comparing the vector of answers obtained by evaluating the questions, to each column of the matrix. If more than one match is found, then ambiguity has been

detected.

It appears that the rule mask method of code generation is preferable to the tree method, only when the quantity of storage used is critical. When a preprocessor is to be implemented, a further problem arises, namely that many high-level languages do not provide bit manipulation. If the answers cannot be optimally encoded, there may not even be a significant saving of storage.

7.1.4 Branch Table

Another code generation technique, which will be referred to as the "branch table" method, has been devised by Veinott [43, 44]. Like the rule mask method, it requires that all the questions in a table be evaluated when the table is executed. As the questions are tested, an index value is accumulated. This value, which identifies the rule selected, is then used to execute an indirect branch to the actions via a branch table. In order to calculate the index value, a power of two is associated with each question: 2^0 for the first question, 2^1 for the second, 2^2 for the third, etc. When a question is tested, a positive answer causes the corresponding power of two to be added to the sum. A negative answer leaves the sum unchanged.

To illustrate this method, consider the following FORTRAN statements, which might be generated for the table

given in section 7.1:

```
      ZZZINX=1
      IF(I.EQ.9)ZZZINX=ZZZINX+1
      IF(P.GE.127.43)ZZZINX=ZZZINX+2
      IF(CNT.LT.100)ZZZINX=ZZZINX+4
      GOTO(10005,10002,10003,10002,10004,10001,10003,
     *     10001),ZZZINX
```

In this code, a computed GOTO statement has been used to perform the indirect branch. Its list of statement numbers acts as the branch table. The actions for rules 1 to 5 would begin with statement numbers 10001 through 10005, respectively. There are duplicates in the statement number list because don't-cares were used to combine rules.

The major flaw in this method is that there must be two to the power N entries in the branch table for a decision table containing N questions. The quantity of storage required for the branch table can be held to a reasonable level by allowing at most four or five questions per table. Extension of this method to tables containing extended answer questions, would be impractical since larger index values would be required and thus the number of branch table entries would be even higher.

The execution of code generated with the branch table method should be faster than that from the rule mask method since the comparison of answers has been eliminated. The tree method remains the fastest since only relevant questions are evaluated, rather than all questions. There

is one other advantage of the branch table method, that is, its simplicity of implementation in a processor.

7.2 Side Effects

The problem of side effects is of interest to the implementor of a decision-table processor since varied results may be produced from the same program when it is translated with different code-generation techniques. By side effect is meant a change which occurs when a statement is executed, in addition to the explicit action specified in the statement. The FORTRAN language has several features which lead to side effects. For example, if the declaration "EQUIVALENCE (I,X)" appears in a program, then the statement "I=I+1" has the insidious effect of changing the value of "X". Although this may be exactly what the programmer intended, it is all to easy to forget that this will happen, especially for someone other than the author.

There are several ways to produce side effects when subprograms are used. These side effects can involve variables in COMMON areas, local variables, and arguments. Since both arithmetical and logical functions may appear in questions, these side effects should be considered in relation to code-generation techniques.

In the first example of code generated with the repetitive testing technique, the number of times each

question is evaluated is dependent on which rule is selected. If, for example, a question contains a reference to a function which is sensitive to the number of times that it is called, then this code may produce unexpected results. It is unclear whether code which evaluates all questions once (rule mask and branch table methods), or code which evaluates only relevant questions (tree method), is preferable.

The simple way of avoiding this particular problem is to rule out the use of function calls in questions. Unfortunately this could be quite inconvenient to the programmer. An alternative course of action is to impress upon programmers the dangers involved with side effects.

CHAPTER 8

Conclusion

Decision tables have been described as a tabular representation of the decisions which must be made, and the actions to be performed, to carry out a procedure. Decision tables are valuable in three areas: first, for recording and communicating the logic involved in a procedure; second, for expressing a procedure in a programming language; and third, for the documentation of computer programs. It was concluded in Chapter 4 that only decision tables are capable of serving all three areas, and that the established techniques - narratives and flowcharts - are distinctly inferior. The main theme of this thesis has been to investigate the second area, that is, the use of decision tables as a control structure in a programming language. The design of a language based on decision tables, and the implementation of this language, have been considered in detail in Chapters 5 and 6. Several algorithms for generating code from decision tables have been examined and criticized in Chapter 7.

Two aspects of decision tables appear particularly inviting for further research. The first is the subject of support software to assist the programmer using a decision-table language. This was discussed in section 4.2 where it was judged that the environment in which a programmer works

plays an important role in determining his productivity. The second aspect is the question of code generation from decision tables. Although substantial research has already been done in this area, it would be unduly pessimistic to claim that no further significant progress can be made. There is one glaring deficiency common to all the code generation techniques known to the author. This is the total failure to consider the actions section of decision tables. If an action is selected on more than one rule of a table, then duplicate copies of the action appear in the generated code. Although this results in code which executes quickly, it may be the cause of a drastic increase in the quantity of storage required. Success in optimizing the use of storage for questions may be completely overshadowed by this inefficiency. Development of algorithms which attempt to optimize both the actions and questions seems highly desirable.

To close this study, two quotations will be presented, the first from Montalbano and the second from Perlis.

"The [decision] table is superior to the flowchart in displaying computer-independent information; the flowchart is superior in displaying computer-dependent information. If the [examples] discussed above were to be programmed for machines which did branching by some other method than binary choice, the flowcharts would be different

but the tables would be unchanged. In this sense, tables are problem-oriented; flowcharts are computer-oriented." [31, pg. 62]

"Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily. Our progress, then, is measured by the balance we achieve between efficiency and generality." [45, pg. 9]

BIBLIOGRAPHY

1. Dijkstra, E.W. "Go To Statement Considered Harmful," Letters to the Editor, Communications of the ACM, XI, No. 3 (March, 1968), 147-148.

2. Leavenworth, B.M. (editor). "Special Issue on Control Structures in Programming Languages," Sigplan Notices, ACM, VII, No. 11 (November, 1972).

3. London, Keith R. Decision Tables. Princeton: Auerbach Publishers Inc., 1972.

4. Pollack, Solomon L., Hicks, Harry T. Jr., Harrison, William J. Decision Tables: Theory and Practice. New York: John Wiley & Sons, Inc., 1971.

5. Cantrell, H.N., King, J., King, F.E.H. "Logic-Structure Tables," Communications of the ACM, IV, No. 6 (June, 1961), 272-275.

6. Nickerson, R.C. "An Engineering Application of Logic-Structure Tables," Communications of the ACM, IV, No. 11 (November, 1961), 516-520.

7. Naur, P. (editor). "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, VI, No. 1 (January, 1963), 1-17.

8. Univac Division of Sperry Rand Corporation. Fundamentals of COBOL Language. UP-7503.1 Rev. 1. U.S.A.: Sperry Rand Corporation, 1968.

9. Kavanagh, T.F. "TABSOL - The Language of Decision Making," Computers and Automation, X, No. 9 (September, 1961), 15, 18-22.

10. McDaniel, Herman (editor). Applications of Decision Tables. Princeton: Brandon/Systems Press, Inc., 1970.

11. Hughes, Marion L., Shank, Richard M., Stein, Elinor Svendsen. Decision Tables. Wayne, Pa.: Management Development Institute, 1968.

12. Denolf, H. "Decision Tables: An Annotated Bibliography," IAG Quarterly, 1, 1968, 67-82.

13. Silberg, Bruce (editor). "Special Issue on Decision Tables," Sigplan Notices, ACM, VI, No. 8 (September, 1971).

14. Gildersleeve, Thomas R. Decision Tables and Their Practical Application in Data Processing. Englewood Cliffs, N.J.: Prentise-Hall, Inc., 1970.

15. McDaniel, Herman. An Introduction to Decision Logic Tables. New York: John Wiley & Sons, Inc., 1968.

16. Van Wijngaarden, A. (editor), Mailloux, B.J., Peck, J.E.L., Koster, C.H.A. Report on the Algorithmic Language ALGOL 68. Report MR101. Amsterdam: Mathematisch Centrum, 1969.

17. Gibson, L., Sutphen, S., Benet, C., Christie, C., (editors). Michigan Terminal System Volume 1: MTS and Computing Services. Edmonton, Alberta: The Universtiy of Alberta Computing Services Department, Rev. Ed., (June, 1972).

18. Weinberg, Gerald M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold Company, 1971.

19. International Business Machines. IBM System/360 FORTRAN IV Language, GC28-6515-7. U.S.A.: International Business Machines, 1968.

20. McCracken, Daniel D., Weinberg, Gerald M. "How to Write a Readable FORTRAN Program," Datamation, XVIII, No. 10 (October, 1972), 73-77.

21. Engel, Frank Jr. "Future FORTRAN Development," Sigplan Notices, ACM, VIII, No. 3 (March, 1973), 4-5.

22. Iverson, K.E. A Programming Language. New York: John Wiley & Sons, Inc., 1962.

23. Griswold, R.E., Poage, J.F., Polansky, I.P. The SNOBOL4 Programming Language. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1971.

24. Cress, P.H., Dirksen, P.H., Graham, J. Wesley. FORTRAN IV WITH WATFOR AND WATFIV. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1970.

25. Cress, P.H., Dirksen, P.H., Ward, S.J. /360 WATFIV Implementation Guide. Waterloo, Ontario: Department of Applied Analysis and Computer Science, University of Waterloo, 1970.

26. Bauer, Henry R., Becker, S., Graham, Susan L., Satterthwaite, E. ALGOL W Language Description. California: Stanford University, Computer Science Department, (September, 1968).

27. Higginson, M. Extensions to ALGOL W. Edmonton, Alberta: Department of Computing Science, University of Alberta, 1972.

28. ASA Sectional Committee X3. "FORTRAN vs. Basic FORTRAN: A Programming Language for Information Processing on Automatic Data Processing Systems," Communications of the ACM, VII, No. 10 (October, 1964), 591-625.

29. Wagner, Robert A. "Common Phrases and Minimum-Space Text Storage," Communications of the ACM, XVI, No. 3 (March, 1973), 148-152.

30. Shapiro, L.P. "Sorting: A Survey, An Analysis and Some Improvements." M.Sc. Thesis, University of Alberta, 1973.

31. Montalbano, Michael. "Tables, Flow charts, and Program Logic," IBM Systems Journal, I, No. 1 (September, 1962), 51-63.

32. Egler, J.F. "A Procedure for Converting Logic Table Conditions into an Efficient Sequence of Test Instructions," Communications of the ACM, VI, No. 9 (September, 1963), 510-514.

33. Montalbano, Michael. Letters to the Editor, Communications of the ACM, VII, No. 1 (January, 1964), 1.

34. Press, L.I. "Conversion of Decision Tables to Computer Programs," Communications of the ACM, VIII, No. 6 (June, 1965), 385-390.

35. Pollack, Solomon L. "Conversion of Limited-Entry Decision Tables to Computer Programs," Communications of the ACM, VIII, No. 11 (November, 1965), 677-682.

36. Sprague, V.G. "On Storage Space of Decision Tables," Letters to the Editor, Communications of the ACM, IX, No. 5 (May, 1966), 319-320.

37. Reinwald, L.T., Soland, R.M. "Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time," Journal of the ACM, XIII, No. 3 (July, 1966), 339-358.

38. Reinwald, L.T., Soland, R.M. "Conversion of Limited-Entry Decision Tables to Optimal Computer Programs II: Minimum Storage Requirement," Journal of the ACM, XIV, No. 4 (October, 1967), 742-755.

39. Kirk, H.W. "Use of Decision Tables in Computer Programming," Communications of the ACM, VIII, No. 1 (January, 1965), 41-43.

40. King, P.J.H. "Conversion of Decision Tables to Computer Programs by Rule Mask Techniques," Communications of the ACM, IX, No. 11 (November, 1966), 796-801.

41. Muthukrishnan, C.R., Rajaraman, V. "On the Conversion of Decision Tables to Computer Programs," Communications of the ACM, XIII, No. 6 (June, 1970), 347-351.

42. Pollack, S.L., Muthukrishnan, C.R. "Comment on the Conversion of Decision Tables to Computer Programs," Communications of the ACM, XIV, No. 1 (January, 1971), 52.

43. Veinott, C.G. "Programming Decision Tables in FORTRAN, COBOL or ALGOL," Communications of the ACM, IX, No. 1 (January, 1966), 31-35.

44. Veinott, C.G. "More on Programming Decision Tables," Letters to the Editor, Communications of the ACM, IX, No. 7 (July, 1966), 485.

45. Perlis, Alan J. "The Synthesis of Algorithmic Systems," Journal of the ACM, XIV, No. 1 (January, 1967), 1-9.

APPENDIX

DET Reference Manual

The following is a reference manual for the programmer using DET, a decision-table programming language. This language is based on FORTRAN, and uses limited answer decision tables as the central control structure. It is assumed that the reader is familiar with both decision tables and the FORTRAN language.

A.1 New Statements

In DET, a decision table is coded using five new statements, each of which is described in detail in the following sections. The five are: TABLE, QUESTION/ANSWER, ACTIONS, ACTION/MAPPING, and EXIT. FORTRAN statements are used for all other processing in a DET program.

A.1.1 TABLE

This statement is coded by entering TABLE somewhere in columns 7 to 72 inclusive. Blanks are ignored by the preprocessor. The statement may not be continued. A FORTRAN statement number may appear in columns one to five. A TABLE statement must not be the last statement of a DO-loop.

The purpose of a TABLE statement is to signal the start of a decision table. Each table must begin with a TABLE statement. A decision table in a DET program is treated as though it were a single FORTRAN executable statement. Execution of a table begins either with a transfer to the TABLE statement, or by "falling into" the TABLE statement.

A.1.2 QUESTION/ANSWER

Each question, plus its answer rules, form a statement. The question precedes the answer rules, and there must be at least one blank to separate them. The question is coded in columns 7 to 72, and may be continued onto successive cards by using the FORTRAN continuation convention. However, even if the question is continued, the answer rules must appear on the first card, following the first part of the question. The answer rules may extend to column 80 but cannot be continued to another card. No QUESTION/ANSWER statement may have a statement number.

The question is a FORTRAN logical expression. The answer rules are two or more contiguous columns containing "Y", "N", or "-". These entries indicate, respectively, yes, no, or don't-care (that the answer to this question is irrelevant in this rule). The same columns must be used for the rules on all questions and actions in a table. These

columns are established by the first QUESTION/ANSWER statement in each table. The answer rules of a statement should not consist entirely of don't-cares since the question would have no bearing on which rule is selected.

Each decision table must contain at least one QUESTION/ANSWER statement. All the QUESTION/ANSWER statements in a table must follow the TABLE statement which began that table.

A.1.3 ACTIONS

This statement is coded by entering ACTIONS somewhere in columns 7 to 72 inclusive. Blanks are ignored by the preprocessor. The statement may not be continued and no statement number is permitted.

The purpose of the ACTIONS statement is to signal the end of the QUESTION/ANSWER section of a table and the beginning of the ACTION/MAPPING section. Each table must contain an ACTIONS statement, which must be placed after the last QUESTION/ANSWER statement.

A.1.4 ACTION/MAPPING

Each action, plus its mapping rules, form a statement. The format of this statement is similar to that for the QUESTION/ANSWER statement. The action precedes the mapping rules, and there must be at least one blank

separating them. The action is coded in columns 7 to 72, and may be continued using the FORTRAN continuation convention. However even if the action is continued, the mapping rules must appear on the first card, following the first part of the action. The mapping rules on all ACTION/MAPPING statements in any given table, must occupy the same columns as did the answer rules in the first QUESTION/ANSWER statement in that table. ACTION/MAPPING statements may not normally have a statement number (an exception is discussed below), and transfers to ACTION/MAPPING statements are never allowed.

The action is any FORTRAN executable statement, subject to the restrictions noted below. The mapping rules contain either an "X" or a blank in each rule. An "X" indicates execution of the action and a blank indicates that the action is to be skipped. Each action should contain at least one "X" in its mapping rules, since otherwise the action will never be executed. If every rule has an "X" then that action will always be executed. In this case the programmer should try to place the action outside the table. However since this is not always possible it is correct to place such a statement within a table.

DO-loops may appear in tables provided that each loop is wholly contained within the action portion of a table. It is also correct for a complete table to appear inside the range of a DO-loop, but DO-loops and tables must

not overlap. Statement numbers are not permitted on ACTION/MAPPING statements, with the single exception that the last statement of a DO-loop must have a statement number. The usual FORTRAN rules concerning the form of a statement number, and the restrictions on the last statement of a DO-loop, apply. Although a statement number is placed on the last statement of a DO-loop, it is not correct to code a transfer (for example, a GOTO) to this label. In FORTRAN programs, one often codes a transfer to the last statement of a DO-loop in order to start the next cycle of the loop immediately, but within a decision table such a transfer must be obtained by the appropriate choice of actions on each rule.

The statement number used by the programmer in a DO statement and on the last statement of that DO-loop, is replaced by the preprocessor in the generated FORTRAN code. This substitution is necessary since the loop may appear more than once in the generated code. Each occurence of the loop is given a unique statement number by the preprocessor. As a result, any reference by the programmer to the original number (such as in a GOTO), will be detected by the FORTRAN compiler as a reference to an undefined statement number.

At least one ACTION/MAPPING statement is required in each table. All ACTION/MAPPING statements in a table must follow the ACTIONS statement in that table.

A.1.5 EXIT

This statement is coded in much the same manner as an ACTION/MAPPING statement. The action must consist of EXIT and cannot be continued. The mapping portion of the statement is optional. If it is present, it must occupy the same columns as the answer portion of the first QUESTION/ANSWER statement. No statement number is permitted.

The purpose of this statement is two-fold: first, it marks the end of a decision table; and second, it provides a method for specifying where control is to be passed after execution of the other actions in each rule. An "X" in a rule on the EXIT statement indicates that the statement following the end of the table should be executed after all the actions on that rule have been executed. Any rule which does not have an "X" on the EXIT statement, must provide for its own exit from the table by selecting an unconditional transfer (or a STOP or RETURN) as the last action on that rule. Each decision table must contain an EXIT statement, which must be placed after the last ACTION/MAPPING statement.

The following are three cases where use of the EXIT statement results in what appear to be violations of the rules stated in previous sections, but which are quite acceptable. First, it is reasonable to see some rules in a

table with an "X" entered only on the EXIT statement. This indicates that under some conditions no actions within the table are to be executed. Second, it can happen that an EXIT statement will contain an "X" in every rule, indicating that after each rule control passes to the statement following the table. Third, when the mapping portion of an EXIT statement is omitted it indicates that all the rules in the table have included an explicit exit. This often occurs when a table is used to pass control to one of several routines depending on some set of conditions.

A.2 Sample Program

The following is an example of a complete program coded in DET:

```
C     THIS DET PROGRAM READS THREE INTEGERS IN 3I10
C     FORMAT, PRINTS THE LARGEST OF THE THREE VALUES AND
C     TERMINATES.
C
      INTEGER A,B,C
      READ(5,1)A,B,C
      TABLE
      A .GT. B        YYNN
      A .GT. C        YN--
      B .GT. C        --YN
      ACTIONS
      WRITE(6,2)A       X
      WRITE(6,2)B         X
      WRITE(6,2)C       X X
      EXIT            XXXX
      STOP
    1 FORMAT(3I10)
    2 FORMAT(' THE LARGEST VALUE IS',I11)
      END
```

In this program, the statements from TABLE through EXIT, inclusive, form one decision table. Those before and after the table, are FORTRAN statements which complete the program but do not involve any decision making.

A.3 Summary of Rules

When coding a decision table there are a number of rules which must be adhered to. The following list is a summary of these rules.

1. The first QUESTION/ANSWER statement must not contain any don't-cares in its answer rules.
2. At most 60 rules may be used on one statement. The answer and mapping rules on each statement in a table must occupy the same columns.
3. A maximum of 25 QUESTION/ANSWER statements may appear in any table.
4. A maximum of 50 ACTION/MAPPING statements may appear in any table.
5. At most 100 continuation cards may appear in any table.
6. The preprocessor generates statement numbers in the range 10000 to 99999. The programmer should therefore use only statement numbers less than 10000. For similar reasons, STOP statements coded by the programmer should have display values less than 10000.
7. No identifiers are generated by the preprocessor. Thus there are no restrictions on the programmer's choice of identifiers.
8. Comments may appear inside tables, even between the cards of a continued statement. These comments will appear in the source listing but are not inserted into the generated FORTRAN code.
9. FORMAT statements may appear in a table, but only between

the ACTIONS and EXIT statements. A FORMAT statement has a statement number but since it is not an action, it does not have mapping rules.

10 Each rule must either have an "X" on the EXIT statement or select a GOTO, STOP or RETURN as the last action of the rule.

11 A table contains a redundancy when the same set of actions is executed regardless of the answer to a question. This situation can usually be avoided by introducing a don't-care.

12 A table is ambiguous when two or more rules can be selected at once. The following example illustrates what should be avoided:

```
YYNN
Y-YN
```

This is ambiguous since when both questions are true, both the first and second rules should be selected. This ambiguity has resulted from mixing don't-cares with yes/no answers.

13 A table is incomplete when for some combination of answers, no rule can be selected. For example, the following questions are incomplete:

```
YYN
YNY
```

An additional rule is required to handle the case where both answers are no.

14 The ordering of the answers is traditionally yes before no, but this is not a requirement. The programmer is free to choose the ordering he prefers. For example, the following cases are equivalent:

```
(1) YYYNN     (2) NNYYY     (3) YYNNY     (4) YYYNN
    YYNYN         NYNYY         YNNYY         NYYNY
    YN---         ---NY         N---Y         -YN--
```

The first two cases each show a consistent ordering, yes before no in the first, and no before yes in the second. The last two cases are valid and equivalent to the first two but have inconsistent ordering which makes them much more difficult to understand. This one objection should be sufficient to force use of a consistent ordering.

[15] IF statements may appear in a DET program either as statements between tables or as actions within tables. However their use (especially in the latter case) is ill-advised. The reason for using DET is to code decisions and actions in tabular structures. This is effected by replacing IF and GOTO statements with decision tables. Some GOTOs will still be necessary to link tables together.

[16] Transfers to statements within a table are not allowed. It is correct, however, to code a transfer to a TABLE statement in order to begin execution of that table.

[17] Transfers from within a table are valid provided they pass control to a statement which is not within a table.

A.4 Use of the Preprocessor

This section describes the commands required to use the preprocessor under MTS. After writing a DET program, this source program is input to the DET preprocessor which translates it into a complete FORTRAN program. The code generated by the preprocessor is compatible with the ANS FORTRAN standard. Following is a prototype of the command used to invoke the preprocessor:

```
$RUN LAFF:DET SCARDS=source SPRINT=listing SPUNCH=fortranpgm
```

The FORTRAN program is referred to as the "intermediate" program. It must be compiled by a FORTRAN compiler before it can be executed. The following is a complete example showing the commands which would be used to process a DET program which is in a file called SAMPLE:

```
$SIGNON csid              'DET SAMPLE RUN'
password
$RUN LAFF:DET SCARDS=SAMPLE SPUNCH=-FTN
$RUN *FORTG SCARDS=-FTN
$IF RC > 4 $SIGNOFF
$RUN -LOAD#
      198       -842       1234                    (data card)
$ENDFILE
$SIGNOFF
```

Both the preprocessor and FORTRAN compiler listings should be checked for error messages. The preprocessor does not attempt to check the FORTRAN code for syntax errors, so many errors will be detected only during the FORTRAN compilation. The FORTRAN code produced by the preprocessor contains comment statements to show the beginning and end of the code generated for each decision table. Thus an error in the FORTRAN listing can easily be traced back to the offending statement in the source program. If the program terminates with a STOP statement which displays a value equal to or greater than 10000, this STOP statement was generated by the preprocessor for one of two reasons: either it detected a rule which did not contain an exit, or there was a serious error in a table, which prevented the preprocessor from generating complete code for that table. The STOP statement caused a halt when the erroneous code was executed. The number displayed with the STOP allows the programmer to locate the table which is in error.

It is possible to save the intermediate code which was ouput by the preprocessor, and modify it to correct

bugs. The advantage in this is that the preprocessing stage can be skipped. However there are disadvantages in that the DET source program will no longer correspond to the final object program, and that the programmer, while making his corrections, will not be coding in DET. Considering the relatively low cost of a preprocessor run, it is advisable not to attempt to fix bugs by modifying the FORTRAN code. Instead, when an error is found, always correct the DET source program and rerun the preprocessor.